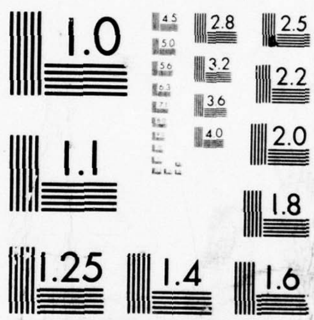
AD-A070 374 KANSAS UNIV LAWRENCE DEPT OF COMPUTER SCIENCE F/G 9/2
IMPLEMENTATION AND EVALUATION OF INTERVAL ARITHMETIC SOFTWARE. --ETC(U)
APR 79 J Q ARNOLD, F P FORD, R G HETHERINGTON DACA39-76-M-0248
UNCLASSIFIED WES-TR-O-79-1 NL

1 OF 1

AD A070374

END
DATE FILMED
7--79
DDC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

TECHNICAL REPORT O-79-1

IMPLEMENTATION AND EVALUATION OF INTERVAL ARITHMETIC SOFTWARE

Report 3

THE HONEYWELL G635 SYSTEM

by

James Q. Arnold, Frank P. Ford, Richard G. Hetherington

Department of Computer Science
University of Kansas
Lawrence, Kansas 66045

April 1979
Report 3 of a Series

Prepared for Office, Chief of Engineers, U. S. Army
Washington, D. C. 20314

Under Contract Nos. DACA39-76-M-0248 and DACA39-77-M-0107

Monitored by Automatic Data Processing Center
U. S. Army Engineer Waterways Experiment Station
P. O. Box 631, Vicksburg, Miss. 39180

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

9 — 1. REPORT NUMBER: Technical Rep[redacted]t 0-79-1

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

6 — 4. TITLE (and Subtitle): IMPLEMENTATION AND EVALUATION OF INTERVAL ARITHMETIC SOFTWARE. Report 3. The Honeywell G635 System.

5. TYPE OF REPORT & PERIOD COVERED: Report 3 of a series

6. PERFORMING ORG. REPORT NUMBER

10 — 7. AUTHOR(s):
James Q. Arnold
Frank P. Ford
Richard G. Hetherington

15 — 8. CONTRACT OR GRANT NUMBER(s): Contract Nos. DACA39-76-M-0248 new DACA39-77-M-0107

9. PERFORMING ORGANIZATION NAME AND ADDRESS:
University of Kansas
Department of Computer Science New
Lawrence, Kans. 66045

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: Integrated Software Research & Development Program, AT11

11. CONTROLLING OFFICE NAME AND ADDRESS:
Office, Chief of Engineers, U. S. Army
Washington, D. C. 20314

11 — 12. REPORT DATE: Apr[redacted]79

13. NUMBER OF PAGES: 88

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):
U. S. Army Engineer Waterways Experiment Station
Automatic Data Processing Center
P. O. Box 631, Vicksburg, Miss. 39180

15. SECURITY CLASS. (of this report): Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

18 WES 19 TR-0-79-1

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

12 93p.

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Algorithms
Computer systems programs
Evaluation
Honeywell G635 System
Interval arithmetic

411239 Du

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This is Report 3 of a series entitled "Implementation and Evaluation of Interval Arithmetic Software." The series concerns implementation and evaluation of an interval arithmetic software package on six different computer systems. The other reports to be published in the series are:

Report 1: The State of the Interval: Evaluation and Recommendations

Report 2: The Honeywell MULTICS System

(Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

79 06 25 006

1

20. ABSTRACT (Continued)

Report 4: The IBM 370, DEC 10, and DEC PDP-11/70 Systems

Report 5: The CDC CYBER 70 System

The most significant fact that has emerged from this project is that interval arithmetic has limited value as a tool for analyzing real algorithms. The limitation is specifically dependency.

The magnitude and complexity of many of the problems being solved at the Waterways Experiment Station make interval analysis of the solution algorithms and sensitivity analysis of the data extremely difficult to accomplish with very high reliability using the current package. The effort to employ interval arithmetic might better be directed toward development of new algorithms based on interval concepts than toward analysis of real algorithms currently in use. Alternatively, redefining the arithmetic operations and interval representation might convert interval into a more practical tool for analysis of the real algorithms.

PREFACE

In December 1975, the Automatic Data Processing (ADP) Center of the U. S. Army Engineer Waterways Experiment Station (WES), Vicksburg, Miss., submitted a proposal to implement and evaluate interval arithmetic, a software system for digital computer numerical analysis, on the Corps of Engineers' primary engineering computer--the WES Honeywell G635. The proposal was later expanded to include the implementation and evaluation of an interval arithmetic software package on six different computer systems. Engineering and scientific data problems were selected to be used on each of the six computers with the interval arithmetic software.

The work was funded by the Office, Chief of Engineers, U. S. Army, through the Integrated Software Research and Development (ISRAD) Program, AT11, Engineering Software Research.

This is Report 3 of a series entitled "Implementation and Evaluation of Interval Arithmetic Software." The other reports to be published in the series are:

Report 1: The State of the Interval: Evaluation and Recommendations

Report 2: The Honeywell MULTICS System

Report 4: The IBM 370, DEC 10, and DEC PDP-11/70 Systems

Report 5: The CDC CYBER 70 System

This report was written by Mr. James Q. Arnold, Mr. Frank P. Ford, and Dr. Richard G. Hetherington of the Department of Computer Science, University of Kansas, Lawrence, Kans. Their work was performed under Contract No. DACA39-76-M-0248, dated 28 April 1976, and Contract No. DACA39-77-M-0107, dated 24 February 1977, and was partially supported by the University of Kansas Computation Center project account and by Research Grant 3564-5038. The work concerned implementation and evaluation of an interval arithmetic software system on the Honeywell G635 computer system.

Dr. J. Michael Yohe, Director of Academic Computer Services, University of Wisconsin-Eau Claire, developed and wrote the interval arithmetic software package which was implemented on each of the six computer systems. Dr. Fred D. Crary, formerly with the U. S. Army Mathematics Research Center, University of Wisconsin-Madison, developed and wrote the AUGMENT precompiler which was implemented on each computer system as a front-end to the interval arithemtic software package. Dr. Yohe and Dr. Crary are specially thanked and recognized for their technical contributions and assistance.

Mr. James B. Cheek, Jr., formerly with the ADP Center, WES, provided initial impetus and guidance for the project. Mr. Fred T. Tracy, ADP Center, WES, provided expert advice and technical guidance during the project. Dr. N. Radhakrishnan, Special Technical Assistant, ADP Center, furnished technical guidance and general project supervision. The project and the report were monitored by Mr. William L. Boyt under the general supervision of Mr. D. L. Neumann, Chief of the ADP Center.

Directors of WES during the project and the preparation of the report were COL G. H. Hilt, CE, and COL J. L. Cannon, CE. Technical Director was Mr. F. R. Brown.

Copies of the other reports of the series, computer listings of the interval program and of AUGMENT for each computer system, and runs of the benchmarks for each computer system may be obtained from the ADP Center, WES.

CONTENTS

CONTENTS

IMPLEMENTATION OF THE 'AUGMENT' PRECOMPILER
AND 'INTERVAL' ON THE HONEYWELL G635

I. PROJECT SUMMARY OF IMPLEMENTATION ACTIVITIES

AUGMENT version 4K was successfully implemented.

INTERVAL II was successfully implemented.

Test programs were run against the combined AUGMENT/INTERVAL II Package.

Some Problems were identified and some suggestions are made.

II. INTRODUCTION

This report describes the Phase I activities of implementing AUGMENT and INTERVAL II on the Honeywell G635. As a result of the University's decision to release the G635 a month earlier than scheduled, Phase I had to be completed on the (compatible) Honeywell 66/60 which replaced the G635. Unfortunately, software errors on both Honeywell systems prevented completion of Phase I on schedule.

Part III of this report contains a discussion of the details of implementing AUGMENT, including a description of the host software errors mentioned above, observations and recommendations regarding efficiency, and arrangements for running AUGMENT. Part IV contains a discussion of the details of implementing INTERVAL II in the format suggested in Dr. Yohe's draft report Implementation of the Package on Other Hardware, June 9, 1976. Part V contains a description of the physical organization of the AUGMENT/INTERVAL II package and information related to use of the package.

Listings of the primitives written at K.U. are given in the Appendices, together with results of testing them. Also, the machine dependent constants needed by INTERVAL II are listed in an Appendix. Finally, a number of program listings together with their outputs are included in the Appendices.

All programs written at the University of Kansas and all modifications to the MRC programming packages to make them operational on the Honeywell systems have been carefully and thoroughly tested, and are believed to be correct. The University of Kansas and the programmers on this project disclaim responsibility for errors that may subsequently arise from use of the programs and systems described herein.

III. IMPLEMENTATION OF AUGMENT

A. DETAILS OF BRINGING AUGMENT UP

The initial effort to bring AUGMENT up was made on the University's G635 which had 200K words (36 bits) for main memory, 380 million characters of disc storage and both 7 and 9 track tape drives. This system supported batch and Time Sharing concurrently. Wherever possible, the time Sharing System was employed in this project in order to utilize on-line editing capabilities. For files the size of AUGMENT this approach was expensive and frequently ran into local boundary limits on file space, processor time, and I/O,...the typical large file problems.

Some minor difficulty was experienced in reading the original tapes sent from MRC. In spite of the obvious needs for standardization, it seems still the case that vendor and installation prerogatives combine to produce a 'tower of Babel' when it comes to exchanging tapes. This was not a large problem, but is noted here for completeness and to alert future users.

1. The Primitives

After studying the documentation, it was decided to rewrite the eight machine dependent primitives PACK, CCODE, MOVHOL, NUMIN, ORDER, STRCHR, STRWDS and STRLNG. These primitives had been written for the G635 at the MRC and were not in the most efficient form. In particular, use of such functions as FLD was frequent, and produced grossly inefficient object code on the G635. All eight primitives had been rewritten and were in the final stages of debugging (only one known error still remained, albeit a vexing one) when the G635 was released and the catastrophic system

errors on the 66/60 halted progress. Subsequently, the versions of these eight primitives which had been sent from Wisconsin were re-installed to eliminate all possible sources of programming errors and this is the version currently being run. The more efficient primitivies will be debugged as soon as possible and installed as a part of the continuing tuning of the systems. A copy of these routines will be made available to WES when they have been thoroughly checked out.

2. MINØ and Variable Dimension (FDARGS)

Two relatively trivial errors appeared in the code sent from MRC. MINO appeared where MINØ was required which was resolved by having MINO call MINØ. Also non-standard (for Honeywell, at least) use of arguments in the routine FDARGS produced a fatal error. Once identified, the correction was easy and was checked with Dr. Crary.

3. Compiler Problems

During the implementation activity on both Honeywell systems, three systems errors were identified. Two of the three are probably exclusive with the new version (SR3I) on the 66/60 and are not errors on version SR8F for the G635. The third error resulted from a basic design flaw and exists on both the G635 and the 66/60 (and all other Honeywell 600, 6000, and 66/XX series systems as far as is known).

a. In outputting error messages, a list of subroutine calls in reverse order is provided by the FORTRAN compiler. Intermittantly, strings of zeros are placed in the name field of the list. In the midst of other chaotic errors, this caused some fruitless searches for array subscript

errors and investigations of table construction algorithms in AUGMENT. Apparently this error does not affect the performance of AUGMENT, and should be ignored if it occurs.

b. A much more serious and disastrously time-consuming system error occured in the computation of arguments to subprograms. After two weeks of constant effort by the entire project team, and aided by frequent long telephone conversations with Dr. Crary in Wisconsin, it was discovered that the 66/60 FORTRAN compiler was mishandling the compilation of statements like

```
PTR = SMMAKE(FSTCOL, ENDCOL, -IABS(CTYP), K, 0, -1, FALSE.)
```

(Statements of this form appear frequently in AUGMENT, the one given being line AUG 38470 in PFETCH). The compiler (version SR2H on the 66/60) produced

```
PTR = SMMAKE(-IABS(FSTCOL), K, 0, -1,.FALSE.)
```

Defining a dummy variable for -IABS(CØL) and assigning it prior to compiling the error producing statement circumvented the problem. This error should not appear in SR8F on the G635.

c. The third error concerns the assembly language instruction Double Precision Floating Compare Magnitude (DFCMG). If DFCMG is used to compare two negative numbers whose magnitudes differ by a large number (for example more than 10^{25}) the return is always "greater than". In effect, the argument in the register is always taken as having magnitude greater than the argument in memory (even though, for example, -1 might be in the register and -10^{30} in memory). This error was first discovered on the G635 but

has subsequently been identified on all large scale Honeywell systems. Furthermore, the single precision form, FCMG, also fails as a result of the same basic design flaw. By Programming, these instructions were avoided. These errors are definitely part of the G635 system and should be publicized locally.

4. Testing the Package

The test programs sent with AUGMENT from the MRC were run successfully with all modifications indicated above.

One difficulty appeared in the way AUGMENT handles the following code:

```
      INTERVAL A
          .
          .
          .
      PRINT 100, A(1), A(2)
```

What happens is

(1) A is not taken over as INTERVAL in the AUGMENT output;

(2) the tables passed to the host FORTRAN are incorrect and produce the fatal error message that A has been previously defined as a scaler.

While it is admitted that the PRINT statement should be

```
      PRINT  100, A ,
```

this is a potentially frequent error. It should produce an AUGMENT message and should not place incorrect information in the AUGMENT output.

B. LINKING

1. Size of the System

In dealing with the space-efficiency balancing problem the size of the program AUGMENT may dictate the need for overlaying. In the present implementation activity, no overlays were used. Instead the emphasis was on efficiency (especially run-time efficiency) and large amounts of file and memory space were employed without particular attention paid to the benefits of overlays. However, in an ongoing-use environment, it is recommended that overlays be employed to reduce costs and space requirements. This will be necessary in a small memory installation; but is desirable in any case. An overlay version is planned for implementation at K.U. and will be available to WES.

2. Function Table Efficiency

The current version of AUGMENT generates function tables for each use. Considerable efficiency might be gained by establishing these function tables once and for all within AUGMENT and reducing the overhead costs to each user.

C. PLAN FOR RUNNING AUGMENT

AUGMENT is a precompiler written in FORTRAN for FORTRAN programs. It should be maintained as an object library on tape or disk. It occupies approximately 600 blocks of disk space. The library includes a mainline and two block common initializing routines called BLDAT1 and BLDAT2. The control cards needed to access the library are:

```
$          IDENT
$          ØPTIØN          FORTRAN, NØMAP
$          FØRTY
$          LIBRARY         LB
$          EXECUTE
$          FILE            LB
$          SYSØUT          21
$          FILE            20

*DESCRIBE

              (description deck)

*BEGIN

              (FORTRAN program)

*END

$             ENDJOB
```

This setup reads a Fortran program and a description deck from file 05 (the usual input file) and outputs these on file 6. File 21 is a copy of the output of AUGMENT which includes errors. File 20 is an S* file which may be used as an input to the normal Fortran compiler.

IV. IMPLEMENTATION OF INTERVAL

Due to the later arrival from MRC of the INTERVAL tape, and at the suggestion of Dr. Yohe, attention was first given to the coding of the machine dependent arithmetic and bounding routines. After reading the documents on INTERVAL it was thought that the extended precision capability of the G635 could be exploited. Since all floating point arithmetic is done in the EAQ registers, one has available a full 72 bit mantissa which would allow guard digit computations. However, a little experience with the Honeywell rounding options was sufficient to demonstrate the pitfalls in that direction. For one thing, the floating store rounded instruction (FSTR) is incorrect in that it is possible to generate a number and its negative, which, after FSTR, do not sum to zero. FSTR always performs an "upward" round. A second problem results from the fact that the exponent has the same size for both single and double precision which precludes use of machine instructions in certain cases treated by INTERVAL. As a result, the machine rounding instructions were abandoned and the arithmetic routine BPAADD, BPASUB, BPAMUL and BPADIV as well as BPACEB (to convert EXTENDED to BPA) and BROUND (to handle directed roundings) were completely written in assembly language.

A. DATA REPRESENTATION

Implicit assumptions that determine choice of data representation are listed by Dr. Yohe as

1. All operations and mathematical functions related to type BPA will be provided explicitly in the BPA package.

2. EXTENDED is used in evaluating the BPA mathematical functions, which requires binding EXTENDED to a higher precision data type than BPA.

3. A complete supporting package including all mathematical functions is required for type EXTENDED.

4. Every BPA number has an exact representation in type EXTENDED.

5. Every FORTRAN integer has an exact representation in type EXTENDED.

6. Conversion from REAL to BPA is exact.

7. Bounds on the accuracy of mathematical functions are available.

In the G635, the data type BPA has been taken as identical to REAL, and EXTENDED has been taken as identical to DOUBLE PRECISION. Although this greatly simplifies the data representation problems, items 1, 3 and 7 above are not readily met as a result of deficiencies in the Honeywell FORTRAN subroutine library and lack of usable documentation for what is there.

The following functions were not supplied in the vendors SR8F FORTRAN for the G635:

BPA (REAL)	EXTENDED (DOUBLE PRECISION)		
TAN	DTAN	DEXP2**	DTANH
CBRT	DARSIN	DSINH	DCBRT
	DARCOS	DCOSH	DLOG2*

* Rename of DLOG

** Rename of DEXP2

In the SR3I version of the 66/60 software, a new and complete set of FORTRAN subroutines are contained in the FORTRAN library. A copy of those routines was requested from Honeywell for delivery with this project. At the time of this writing it appears likely that the request will be granted. This may be handled at WES by installing the new library in place of the current one or by treating it as a special library for use with the AUGMENT/INTERVAL package. In any event, if the complete FORTRAN library is not used, the G635 FORTRAN library will not support INTERVAL unless locally developed subroutines are made available.

B. CODING OF PRIMITIVES

This section contains a description of the algorithms developed for the BPA arithmetic and for BROUND, together with explanations of the decisions made. Initially these algorithms were written to incorporate special handling for the asymmetric numbers described below. However, after having obtained a working version which allowed the special cases to be treated correctly as far as INTERVAL was concerned, it was decided that these special cases caused inefficiencies and difficulties of understanding beyond their usefulness. For this reason a change was made to the algorithms which eliminates the special cases as either inputs to, or results from the arithmetic routines. The current versions contain these modifications as described below. However, without these cases, the requirement for passing arguments with the correct sign to BROUND is no longer justified. This in turn, implies that some improvement in efficiency will be realized by

a return to end sign correction as part of BROUND (this had been abandoned when it was observed that two passes through BROUND might be required for the special cases mentioned above) rewriting all the primitives in order to accomplsih this increase in efficiency is a possible step in tuning the system.

Normalized, floating point, two's-complement arithmetic on the Honeywell 635 dones not have symetric bounds on the numbers it can represent. For single precision, normalized numbers, the following ranges are given;

<u>negative</u> $-(1+2^{-26})(2^{-129}) \geq N \geq -2^{129}$

<u>positive</u> $2^{-129} \leq N \leq (1-2^{-27})(2^{127})$

Furthermore, zero is represented as $(0)(2^{-128})$.

Two main problems are caused by this lack of symmetry: the absolute value of the negative number with the largest magnitude cannot be represented, and the smallest, normalized, positive number has no normalized negative counterpart (although its negative can be given by an un-normalized number). To preserve the accuracy of the arithmetic done by INTERVAL, we must test for and identify these cases at various points in the computation.

Since the negative number with the largest magnitude does not have a negative, we do not allow it to be either an argument to or an answer from any of the arithmetic routines. If that number is passed to any of the arithmetic routines as an argument, the overflow indicator is set, the most negative number allowed by the arithmetic routines is loaded into the A register of Interval, and control is passed to BROUND to set the correct fault indicator (overflow or infinity) depending on the brounding option specified for that particular operation. If the negative

number with the largest magnitude is computed in any of the arithmetic routines, then it is treated like an overflow condition, which it actually is to Interval - even though it may not be an overflow to the computer.

Although there is a negative number with the same magnitude as the smallest positive number, it is not normalized, and therefore it cannot be returned as an answer from the arithmetic routines. Consequently, it is not allowed as an argument either. If the smallest, positive, normalized number comes in as an argument to the arithmetic routines, the exponent underflow indicator is turned on, and control passes to BROUND. In BROUND, it is treated as a case of underflow by one.

All arithmetic routines pass a number with the correct sign to BROUND. Corrections for the residue indicator also take place in the arithmetic routines.

```
          //In these arithmetic routines, MINNEG is the negative
          number with the largest magnitude, which is representable
          in the computer. NEGBND is the negative number with the
          largest number which is allowed in Interval. MINPOS is
          the smallest, normalized positive number which can be
          represented. Additional comments are given for numbered
          lines.//
BPASUB:   //compute  A ← A1 - A2 by negating A2 and adding //
          U ← A2
          if U = 0
               then A ← A1
                    return
```

```
            else U ← -U
                 goto SUBIN             //go to add routine to add A1,A2//
        fi
BPAADD: //compute A ← A1 + A2//
        U ← A2
        if U = 0                        //if U = 0, then A1 is the answer//
            then A ← A1
                 return
        fi
SUBIN:  A ← A1
        if A = 0                        //if A = 0, then the answer is in U//
            then A ← U
                 return
        fi
1       if |U| < |A|                    //put argument with smaller//
            then  U↔A                   //magnitude in A//
        fi
        if A < 0                        //A must be made positive//
            then A ← -A
                 U ← -U
                 SC ← 1
        fi
        TEMP ← E(U) - E(A)              //we compute the shift count//
2       if TEMP > 0                     //if TEMP ≤ 0, we need no shift//
3           then E(A) ← E(U)
4                shift M(A) right TEMP bits          //line up mantissas//
        fi
```

```
        A ← A + U
        if SC = 1
            then if EO = 1
5                   then A ← -A
6                        EO ← 1
7                        EU ← 0
                    else if EU = 1
8                           then A ← -A
9                                EU ← 1
10                               EO ← 0
                            else A ← -A
                         fi
                fi
        fi
        if RI = 0                              //if no residue and M(A) = 0//
            then if A = 0                      //then we don't need brounding//
                    then return
                    else goto BROUND
                 fi
        fi
        //now we need a residue correction//
11      if SC = 0
12          then if M(A)_{28-63} = 0
13                  then M(A)_{63} ← 1
                 fi
                 goto BROUND
14          else if M(A)_{28-63} = 0
```

```
15                    then M(A) ← M(A) - (0...01)
                fi
                goto BROUND
           fi                                    //end of BPAADD//
```

Additional comments:

<u>lines</u>

1 At the present time (1976) the DFCMG instruction does not work on the Honeywell 635; consequently, we must work around it, using other instructions to do the same thing. We have done this by making both of the arguments positive, comparing them (which is the same as comparing their magnitudes), and then putting the smaller one in A and the bigger one in U with their correct signs. The problem with the DFCMG was not known by Honeywell at the time it was reported to them.

2-4 E(U) - E(A) will be negative in one special case: M(A) = -M(U) = 010...0 = ½. A negative number normalizes to have a 1 in the first bit of the mantissa; consequently, its exponent is one less than the exponent of its absolute value. In this case, we do not want to shift the exponents. When the difference of the exponents is zero, even when one is negative and one is positive, we do not need to shift the mantissas, because they line up already. Otherwise, we must shift the mantissa of the A register right, to line it up with the mantissa of the U register. The number of bits we must shift it is the difference, E(U)-E(A).

5-10 After the addition of the two arguments, we must restore the sign of the answer. If we produced an overflow in the addition, simple negation will produce the mantissa with the correct sign. One possible answer for the overflow is the smallest positive normalized number. If we negate that, we will get an underflow, and so after the negation, we must be sure to turn off the EU indicator, and to turn the EO indicator back on.

Furthermore, if we produced an underflow in the addition, one possible answer is the negative number with the largest magnitude representable on the machine. If we try to negate that, we will get the correct mantissa, but we will also produce an overflow because that number does not have an absolute value which is machine representable. Once again, after we do the negation, we must be sure to turn the EO indicator back off (if it was turned on), and we must be sure to reset the EU indicator.

11-15 At this point we need to make a residue correction, and we have four cases to deal with:

1. $SC = 0$ and $A \geq 0$
2. $SC = 0$ and $A < 0$
3. $SC = 1$ and $A \geq 0$
4. $SC = 1$ and $A < 0$

Case 1. Both A1 and A2 were positive. Thus the bits lost decreased the magnitude of the answer. If any of the bits $M(A)_{28-63}$ are 1, the correction has already been made, so only if

$M(A)_{28-63} = 0$ do we set $M(A)_{63} \leftarrow 1$.

Case 2. A1 and A2 had different signs, and the positive had the smaller magnitude. Thus the bits lost increased the magnitude of the answer. If $M(A)_{28-63} = 0$, then we must set $M(A)_{63} \leftarrow 1$ to correct for the gain in magnitude. If $M(A)_{28-63} \neq 0$, however, the correction has been made.

Thus, Case 1 and Case 2 require identical treatment.

Case 3. A1 and A2 again differed in sign, but this time the positive argument had the larger magnitude. Therefore, the bits lost increased the magnitude of the answer. If $M(A)_{28-63} \neq 0$ and $A > 0$, any truncation of non-zero bits in $M(A)_{28-63}$ will insure proper brounding when augmentation is implied. If $M(A)_{28-63} = 0$, however, truncation will leave the answer too large, and so we must subtract 0...01 from M(A) to insure proper brounding when truncation is implied.

Case 4. Both A1 and A2 were negative, and thus the non-zero bits lost decreased the magnitude of the answer. If $M(A)_{28-63} \neq 0$, then truncation will insure correct augmentation, while the non-zero bits will guarantee proper rounding towards zero. But once again, if $M(A)_{28-63} = 0$, neither augmentation nor

brounding toward zero will work, and so we must subtract 0...01 from M(A).

Thus Cases 3 and 4 require identical treatment.

```
BPAMUL:  //compute A←A1*A2//
         //convert arguments A1,A2 to positive values.  Set SC:=1
           if answer must be sign - corrected before bround//
     if A1 < 0
          then A1←-A1
               SC←1
     fi
     if  A2<0
         then A2←-A2
              SC←1-SC
     fi

         //compute A1*A2 without normalizing//
           A←A1*A2
         //handle overflow or underflow from above//
     if exponent overflow then go to BPAMEØ fi
     if exponent underflow then go to BPAMEU fi
         A←normalized A
     if exponent underflow from normalization then go to BPAMEU fi
     if SC=1 then A←-Afi
     go to BROUND

BPAMEU:  A←-A
         ØPTIØN←rounding option
         BPAFLT ← 0
```

```
        if SC=1 then A←-A fi
        go to .EU.
        //.EU. is a section of BRØUND for exponent underflow//
BPAMEØ: ØPTIØN ← rounding option
        BPAFLT ←0
        //normalize A.  This may fix exponent overflow//
          E(A)=0
        if SC=1 then A ←-A fi
          A←normalized A
        //E(A) has shift amount//
          E(A) = E(A) + E(A1) + E(A2)
        if exponent overflow then go to .EØ. fi
BPADIV: //compute A←A1/A2//
        //convert to positive values//
        if A1 < 0 then
           A1←-A1
           SC←1
        fi
        if A2=0 then go to ZERØDV fi
        if A2 <0 then
           A2←-A2
           SC←1-SC fi
        //determine if dividend larger than divisor.  This section
          (specially DVF instruction) works only if dividend ≤ divisor.//
        if M(A1)>M(A2) then
           M(A) ← M(A) /2
           SHIFT=1
        fi
```

```
        if M(A1) = M(A2)
           SHIFT = 0
        fi
    A← quotient (M(A1)/M(A2))
    Q← remainder (M(A1)/M(A2))
    if Q ≠ 0 then RI:=1 fi
    //RI is residue indicator//
    //negate if necessary.  Clear exponent to avoid its interference//
      E(A)=0
    if SC = 1 then A←-A fi
    //now compute exponent. (A) is from normalization//
      E(A) = SHIFT
      A ← normalized A
      E(A) = E(A) + E(A1) - E(A2)
    if E(A) > 128 then go to EXPØ fi
    if E(A) <-128 then go to EXPU fi
    if RI = 1 then M(A)= M(A)+1 fi
EXPØ:    ØPTIØN ←0
         BPAFLT ←0
         go to .EØ.
EXPU:    ØPTIØN ← rounding option
         BPAFLT ←0
         go to .EU.
ZERODV:  BPAFLT ←4
         go to EXIT
```

Understanding the brounding of two's complement numbers requires some familiarity with the way those numbers are represented, particularly, the negative numbers.

A positive, single-precision, normalized, floating-point number in the Honeywell 635 must have a 0 in bit zero, and a 1 in bit one. After that, it may have any combination of 1's and 0's. Consequently, the mantissa may represent a number as small as 2^{-1}, and as large as $1-2^{-27}$. If we add bits to a positive mantissa or concatenate bits at the right end of a positive mantissa, and those bits are nonzero, then we increase the magnitude of the mantissa; if we drop a non-zero bit off the right end of a positive mantissa, we will decrease the magnitude of the number represented by the mantissa.

The Honeywell 635 has a 28-bit single-precision mantissa, and a 64-bit double-precision mantissa. The rounding options for Interval are accomplished as follows:

U,A: If any of the bits in $M(A)_{28-63}$ are non-zero, then we want to return the next higher single-precision mantissa as the answer. If $M(A)_{28-63} = 0$, though, we don't want to change the answer we have. Thus we want to add a number to A that will propagate a carry from any of the bits 28-63 into bit 27 of the mantissa without generating a carry if $M(A)_{28} = 0$.

L,T: Any non-zero bits in $M(A)_{28-63}$ make the answer too large, so we truncate them and use only the single precision answer. If the extra precision word is all zero, then we may simply truncate in that case too.

R: If we are rounding, a 1 in $M(A)_{28}$ means that we round

up, while a 0 means we round down. Thus, we may add a one into $M(A)_{28}$ and we will get the correct answer. If there is a one there, we will generate a carry into the single precision word, if there is a zero, then no carry will be generated, and the answer will be rounded down as desired.

Negative numbers are somewhat different. First, they normalize differently; they must have a 1 in the first bit, a zero in the second bit, and after that, any combination of 1's and 0's may occur. Thus, the range of negative mantissas is from -1 to $-(2^{-1} + 2^{-27})$. A negative mantissa may represent a value with a greater magnitude than a positive one, but a positive mantissa may represent a value with a smaller magnitude than a negative one. If we add bits to a negative mantissa or concatenate bits at the right end of a negative mantissa and those bits are non-zero, we will decrease its magnitude; if we chop a non-zero bit off the right end of a negative mantissa, we will increase the magnitude of the number represented by the mantissa. This is very important to our brounding. The brounding options are executed as follows:

U,T: Both U and T imply brounding toward zero; if any of the bits in $M(A)_{28-63}$ are one, then we want to generate a carry into $M(A)_{0-27}$ to decrease the magnitude of the answer. Only if $M(A)_{28-63} = 0$ will we not want to generate that carry. Thus we may add the same constant here as we did for augmentation in the positive case.

L,A: To find the lower bound, any ones in $M(A)_{28-63}$ must be ignored, because they decrease the magnitude of

the answer, and to find the lower bound (or to augment) we must increase the magnitude. If there are no 1's there, we have the correct answer. Thus for both L and A options here, we may simply chop off the double precision word.

R: If we are rounding a negative number, then we want to go away from zero if the extra precision word has a 1 in bit 28 and 0's in $M(A)_{29-63}$ or if bit 28 has a zero in it. To take the answer away from zero, we want to truncate those bits. On the other hand, if $M(A)_{28} = 1$, and any of $M(A)_{29-63} = 1$, then we want to bround toward zero. Consequently, we want to generate a carry into $M(A)_{27}$ of $M(A)_{28} = 1$ and any of the bits in $M(A)_{29-63} = 1$.

One important thing to realize about this brounding is that overflow and underflow may occur in certain conditions. We may get an overflow when augmenting positive numbers, and we may get an underflow by one if we truncate negative numbers. These conditions must be checked when the brounding is done, and the appropriate steps must be taken if a fault is recognized.

Three numbers are used to do brounding: UTA is used for the U option (positive and negative), the T option (negative), and the A option (positive); NEGRND is used by the R option for negative numbers; POSRND is used by the R option for positive numbers. The mantissas of the three numbers are presented here.

```
UTA     | 00...00 | 11...11 |
          0     27  28    63

NEGRND  | 00...00 | 011...11 |
          0     27  28     63

POSRND  | 00...00 | 100...00 |
          0     27  28     63
```

When one of these numbers is used, its exponent is set to the exponent of the number being brounded.

Finally, we may present the following decision table for the brounding routine.

OPTION	SIGN OF ANSWER +	SIGN OF ANSWER -
U	add UTA	add UTA
L	none	none
T	none	add UTA
A	add UTA	none
R	add POSRND	add NEGRND

In the following brounding routine, several variables are used, and they represent certain floating-point constants.

MINNEG is the most negative machine representable number.

NEGBND is the most negative number allowed as an argument to or as an answer from the arithmetic routines.

MAXNEG is the negative, single-precision, normalized number closest to zero.

MINPOS is the smallest, normalized, positive number which can be represented in the machine.

POSBND is the smallest, single precision, normalized number allowed as an argument or an answer from the

arithmetic routines.

MAXPOS is the largest, positive number.

UTA, NEGRND, and POSRND are as described.

```
BROUND: normalize A
        if CO ≠ 'U' and CO ≠ 'L' and CO ≠ 'T' and CO ≠ 'A' and CO ≠ 'R'
            then CO ← 'R'              //if correction option is invalid, use R//
        fi
        case
         :EO = 1:
            if A ≥ 0
                then A ← MAXPOS
                    if CO = 'A' or CO = 'U' or CO = 'R'
                        then INF ← 1
                    fi
                            return
                else A ← NEGBND
                    if CO = 'A' or CO = 'L' or CO = 'R'
                        then INF ← 1
                    fi
                            return
            fi
         :EU = 1:
            if  A ≥ 0
                then if CO = 'A' or CO = 'U' or (CO = 'R' and underflow by 1)
                        then A ← POSBND
                        else A ← 0
                    fi
                    return
```

```
        else if CO = 'A' or CO = 'L' or (CO = 'R' and underflow by 1)
              then A ← MAXNEG
              else A ← 0
           fi
           return
   fi
:CO = 'U':                        // find the upper bound on A//
   A ← A + UTA
   if EO = 1                      //overflow may occur when A is positive//
        then INF ← 1
           A ← MAXPOS
        else if EU = 1            //underflow may occur when A is negative//
              then A ← 0
           fi
   fi
   return
:CO = 'L':                         //find lower bound//
   return                          //store single-precision in all cases//
:CO = 'T':                        //bround toward zero//
   if A ≥ 0                       //store single-precision if positive//
        then return
        else A ← A + UTA          //A < 0//
           if EU = 1
              then A ← 0
           fi
           return
   fi
:CO = 'R':                        //round to nearest machine number//
   if A ≥ 0
```

```
                    then A ← A + POSRND
                         if EO = 1
                              then A ← MAXPOS
                                   INF ← 1
                         fi
                         return
                    else A ← A + NEGRND                    //A < 0//
                         if EU = 1
                              then A ← MAXNEG
                         fi
                         return
               fi
          :CO = 'A':                                       //augment A//
               if A ≥ 0
                    then A ← A + UTA
                         if EO = 1
                              then A ← MAXPOS
                                   INF ← 1
                         fi
                         return
                    else return                            //A < 0//
               fi
          end
     //end of bround//
```

C. MACHINE DEPENDENT CONSTANTS

The machine dependent constants, given in BPA format, which must be provided to INTERVAL are listed in [1]. These constants are located in COMMON/BPACON/ and for the G635 version the actual numbers used are given in Appendix B. While many of these required constants are obtained in a straightforward manner (e.g. the left endpoint of the INTERVAL containing PI), several must be derived from vendor documentation (e.g. half-length of INTERVAL about 1 where LN accuracy decreases). The approach taken in determining these constants was the pragmatic one of comparing output from the routines with published tables [2,3,4,5,6]. It is expected that some improvement in the INTERVAL results might be realized by spending more time on the evaluation of these constants.

One additional comment in passing: the description of the constant FRACBD does not clearly bring out the fact that FRACBD is that constant which when added to any number effectively truncates the decimal part and produces a real integer.

D. ERROR BOUNDS FOR EXTENDED PRECISION ROUTINES

As described in [1], the method of implementing interval mathematical functions is to assign an error term to each function which is then utilized in the obvious way to determine the value (interval) taken by the function. Let f be a continuous interval valued function of an interval variable which we write as

$$f([a,b]) = [c,d].$$

Then, there exist points a' and b' in [a,b] such that f(a') = c and f(b') = d. If the EXTENDED library routine (double precision routine in the present instance) produces the value f_E with an error bound ϵ, then the result computed by INTERVAL II is

$$\hat{f}([a,b]) = [\nabla(f_E\ (a')-\epsilon,\ \Delta(f_E\ (b')+\epsilon)]$$

where ∇ and Δ are downward and upward directed roundings, respectively.

From this formulation, it is seen that accurate error bounds are essential to optimize interval function evaluations. However, it is extremely difficult to calculate these bounds accurately and hence the conservative approach suggested by Dr. Yohe in [1] has been adopted. In effect this means using the error bounds specified in the vendor documentation for these routines.

E. MODIFICATIONS TO INTRAP

INTRAP provides a printed record of faults encountered in interval operations. It is assumed that each routine which can call INTRAP has at most three arguments, and that the following information is provided to INTRAP:

ID	a 3-character suffix specifying the calling routine
TYPA	the type of argument A
TYPB	the type of argument B
TYPR	the type of the result

Values of TYPA, TYPB, TYPR may be any one of

0 null

1 FORTRAN INTEGER

2 FORTRAN REAL

3 FORTRAN DOUBLE PRECISION

4 BPA

5 EXTENDED

6 INTERVAL

Because BPA was identified with REAL and EXTENDED was identified with DOUBLE PRECISION, the built-in conversion routines were suitable for this implementation and no new code was added to INTRAP.

F. PROCESSING WITH AUGMENT

INTERVAL consists of a description deck for use with AUGMENT and a package of compatible subroutines and functions to handle INTERVAL data. It should be maintained as two files--a library file and a description deck. The control

cards needed for it would be:

```
$    IDENT
     .
     .  (AUGMENT CONTROL CARDS)
     .
$    FILE      20, AISD
* DESCRIBE
$    SELECT    INTERVAL DECK
* BEGIN
     Fortran program
* END
$    OPTION    FORTRAN, NOMAP
$    FORTY
$    LIBRARY   LC
$    FILE      S*,AIDD
$    EXECUTE
$    PRMFL     LC, INTERVAL LIB.
     Fortran data
$    ENDJOB
```

G. CHECKING THE PACKAGE

The test decks supplied by Dr. Yohe were run against INTERVAL II in the manner described in the preceding section, and the output was checked with the sample output. As mentioned in section E, these tests utilized INTRAP as it was sent to us by MRC. The machine dependent constants described in C and D were used in these tests. The results matched and the test was assumed passed. Appendix C contains these test results.

H. TUNING THE PACKAGE

Tuning the package is interpreted to mean making any modifications to code or data which will result in reducing time or space requirements, or will produce more accurate interval results. (Conflicts may occur between these efforts; for example, run time may be increased in order to get tighter intervals.) The major activities planned for tuning the system are:

1. Replacing AUGMENT Primivitives discussed in section IIIA.1. of this report.
2. Constructing an overlayed version of the package. (B-1 above)
3. Fixing the AUGMENT function tables rather than regenerating them for each use. (B-2 above)
4. Improve error bounds for extended precision routines (C and D above)
5. Remove all unnecessary calls in the AUGMENT output (as suggested in [1])
6. Rewrite arithmetic primitives for INTERVAL as discussed in III.B of this report.

V. USING THE COMBINED AUGMENT/INTERVAL PACKAGE

A. ORGANIZATION (PHYSICAL DESCRIPTION)

The particular organization to be used at WES will depend to great extent on local decisions. The modules being delivered include

1. Object library of AUGMENT programs.
2. Source library of AUGMENT programs.
3. Object library of INTERVAL programs.
4. INTERVAL description deck for AUGMENT.

A perm file containing control cards (see III.C) needed to load and execute the AUGMENT object library and a perm file containing control cards (see IV.F) to run AUGMENT and the description deck will be created and accessed by $ SELECT cards. Another perm file to describe the necessary libraries and the output file from AUGMENT to permit the FORTY compiler to execute the AUGMENT output will also be created and accessed by a $ SELECT card.

B. USER INFORMATION

For input/output, it should be noted that a variable of type INTERVAL is converted to a variable of type REAL with dimension 2. Therefore, normal FORTRAN I/O methods may be used.

Section IV F contains specific control card information for running programs.

EVALUATION OF INTERVAL ARITHMETIC
FOR THE HONEYWELL G635

VI. INTRODUCTION

At its instigation this project was viewed as a relatively routine implementation of the Augument/Interval programming system on the GE/ Honeywell 635 computer followed by a set of benchmark runs to investigate the efficacy of using Interval Arithmetic to analyze error in the problem solution methods for use at WES. From the beginning, difficulties were encountered in all aspects of the project activity, including operating system failures and misunderstood communications at K.U.; delays in delivery of tapes, inefficiencies, and occassional errors in coding sent by MRC; and untested benchmark programs and data sent by WES. While some difficulties of this sort are to be expected, the sum total of all of them coupled with the rather poor performance of the initial untuned Interval/Augument system lead to extensive delays in the completion of the contracted work. Every effort has been made to satisfy the letter and the spirit of the contractual agreement on the parts of all parties involved. However, the simple fact is that the task was monumentally greater than anticipated and even now leaves many areas of potentially fruitful further investigation unexplored.

Briefly, the chronology of events since the report on implementation given November 1976 at Vicksburg is:

Corrected versions of the benchmark algorithms were received	Nov. - Dec. 1976
Overlay version of AUGMENT/INTERVAL implemented	January 1977
GAUSSE benchmark runs initiated after receiving the complete data	February 1977
SPLINE benchmark run successfully	March 1977

Report on completion of Phase III sent to WES	March 1977
FFT benchmark runs initiated	April 1977
INTERVAL arithmetic and brounding routines rewritten	May - June 1977
BPA subroutine calls eliminated	July 1977
Reruns of benchmarks for timing comparisons	July 1977

VII. DISCUSSION OF DIFFICULTIES ENCOUNTERED IN USING INTERVAL ARITHMETIC

A. DEPENDENCY

There is inherent in the operations of interval arithmetic a defect which may increase width of intervals in computed results unrealistically. The generic term describing this defect is dependency. It derives from endpoint calculations which make use of expressions which are dependent, but the interval arithmetic operations treat them as independent. Thus, if $Y = \frac{X}{X}$, where $X = [1/2,2]$, the definition of interval division produces $Y = [\frac{1}{4},4]$. This result includes the desired (correct) answer [1,1] but at the same time has increased the interval width substantially. Other anomolies include the fact that additive and multiplicative inverses don't exist in the expected forms: if $X = [a,b]$, $a \leq b$, then $X - X = [a - b, b - a]$ which is not [0,0] (unless $a = b$) and $X/X = [\frac{a}{b}, \frac{b}{a}] \neq [1,1]$ (unless $a = b \neq 0$). The problem is that the two occurrences of the same interval X, in these illustrations, are treated by the interval arithmetic operations as independent when in fact they are dependent. There is no simple resolution to this difficulty because the definitions of the interval-arithmetic operations must treat the two interval operands as independent in order to guarantee inclusion of the correct result in the interval answer. Program testing of the operands for dependency is hopelessly complicated in any but the simplest sequences of operations. The fact is that interval arithmetic incorporates an error generation characteristic which is distinct from ordinary arithmetic error considerations. The result is that proven methods and algorithms of real arithmetic often fare badly when converted directly into interval arithmetic.

The dependency problem is most effectively dealt with by avoidance - simply not applying interval analysis to those algorithms or data which are known to produce large dependency error. Unfortunately, there do not yet exist practical methods for deciding, before the fact, what will be the efficacious interval algorithms and what will not. Experimentation often is the most direct means of obtaining such information although some guidelines have been developed by F.N. Ris [13] and E.R. Hansen [12]. The upshot of this situation for an installation such as WES is that many production programs already in use cannot really be effectively analyzed as interval arithmetic programs. In those instances where 'good' results are obtained from the interval runs, one can be reasonably sure that the real arithmetic algorithm is numerically stable. But when interval results are meaningless because the intervals are too wide, then one cannot conclude that the real arithmetic algorithm was also unstable - only that the increase in interval widths was too great. If the dependency width (a term used by F.N. Ris [13] to describe that part of the increased interval width due exclusively to dependency) is the principal contributor to the increased widths of interval results, then the algorithm should not be analyzed using interval arithmetic and a different analysis technique should be employed. On the other hand, if dependency width is small, even though the interval width of results is too large, the algorithm may be salvaged by some one or more of the traditional real arithmetic error reduction procedures, e.g., using higher precision calculations, more terms in the approximations, etc. Of course, such modifications to an algorithm may cause it to exhibit dependency which it previously did not have. To illustrate, consider the simple series

$$f(x) = -1 + x - x^2 + x^3 - x^4 \ldots$$

If the first two terms are used, there is no dependency problem. But should two terms provide insufficient accuracy, adding the third term will improve accuracy and at the same time introduce a dependency problem.

B. COMPARISON OF INTERVALS

Dependency is by far the overriding consideration in using interval arithmetic. If significant in an algorithm, it may obliterate any meaningful interval results. However, other sources of difficulty may enter the interval arithmetic operations for a particular solution method. One of these is the problem encountered in translating the compare operation from real arithmetic to interval arithmetic. An intuitive approach might be to compare endpoints and when the sup of one interval is less than the inf of the other interval, then say the first interval is less (smaller) than the second. This works fine for non-overlapping intervals. When the intervals overlap, an appeal might be made to a real comparison of their midpoints. But saying that the interval $[\frac{12}{25}, \frac{51}{100}]$ is less than the interval [0,1], as would follow, leads to other difficulties since nearly half the numbers in [0,1] are less than all values in $[\frac{12}{25}, \frac{51}{100}]$. The basic problem is that while real numbers are completely ordered, intervals are not. The result is that no entirely satisfactory way of translating a comparison of real numbers into a comparison of interval numbers exists.

C. ZEROS ENTERING COMPUTED INTERVALS

Yet another source of difficulty in interval arithmetic stems from the fact that as a calculation progresses, the widths of the intermediate

interval results generally tend to become larger. In particular, to guarantee the correct (real) answer is captured within the final interval, it is necessary to round intermediate answers to machine representable numbers which assure this inclusion. While this strategy is obvious and correct, the effect is to expand interval width. In most cases this somewhat conservative approach is defensible and does not lead to problems. But when, in the course of a calculation, the intervals become null (contain the real number zero), or overlap, otherwise innocent operations may produce catastrophic interval growth.

D. TIMING CONSIDERATIONS

Timing and space considerations can be viewed in a general way as well as particularizing them to a certain application program. In this paragraph we will comment in general on these matters. In the next section of this report more specific details of timing will be presented for each benchmark. The efficiencies gained from tuning the package will be presented in the section following that.

Because the current version of AUGMENT/INTERVAL is designed for generality and portability, there are many inefficiencies in the package. Apart from the obvious increases in memory required for interval over real arithmetic, there are the time and space costs of generating function tables in AUGMENT for each use. Modifying AUGMENT to eliminate function table generation with each use was investigated and found to be a substantial systems programming effort. However, in an environment where there is frequent production use of AUGMENT, the effort to do the modifications would likely be justified. This change has not been made in the current system.

Another inefficiency caused by the desire for generality is the multiple levels of subroutine calls involved in even the simplest source language statements. For example, the statement X = A * B, for A, B, X interval variables, generates a total of 134 subroutine calls, nested 3 deep.

Because in the H635 version BPA is identified with REAL and EXTENDED with DOUBLE PRECISION, we were able to reduce this sort of inefficiency by inserting replacement statements for subroutine calls. This work is described in Section IV of this report. Further effort along the lines of removing the nested subroutine calls could prove very beneficial to overall efficiency of the package.

Finally, some discussion with a faculty colleague has concerned the hardware implementation of interval arithmetic. There is no doubt that significant improvements in run times can be achieved by replacing the arithmetic primitives with hardware. However, dependency problems will not be rectified by hardware implementation.

VIII. THE ENGINEERING ALGORITHMS

A. SPLINE

A report made March 15, 1977 contains the results of the benchmark run of the subroutine SPLINE. These results are also included in this final report for the sake of completeness. As reported earlier, SPLINE ran under INTERVAL with relative ease. The interval widths of the answers varied from 0.0003 to 0.0012 and the intervals were very nearly centered on the real results. Thus the SPLINE algorithm seems to be fairly stable when performed in interval arithmetic and as remarked in Section II of this report, this indicates stability of the algorithm under real arithmetic. The timing factor was 13.4, that is, it took 13.4 times as long to run under AUGMENT/INTERVAL as it did to run in real arithmetic. After tuning the system, as described in Section IV of this report, the same run was repeated with the result that the intervals were generally narrower and the timing factor was reduced to 12.2, a 10% improvement. Listings of these runs are included in Appendix B.

B. FFT

Initial efforts to run the FFT benchmark failed as a result of an error in the program modifications written at WES. When this problem was finally ferreted out in June, the FFT results were still not very good. Closer examination revealed an error in the INTERVAL cosine routine sent to us by MRC. After rewriting INTCOS we discovered that the same fix had been distributed by MRC last summer and had been added to our working system. It is not clear how the correction was lost, nor whether

the uncorrected version was included in the package delivered to WES in November. However, the recently delivered system contains the corrected INTCOS.

Because of the difficulties mentioned above, a smaller problem had been constructed at K.U. to test FFT. It consists of an 8-point approximation to sin $\frac{\pi X}{2}$ over the interval [0,2]. Interval results for this test appeared at first to have failed to capture the real answer, that is, the 'correct' answers were not in the intervals computed. After careful analysis of the algorithm and the programs, it was determined that the problem was in the real algorithm. The intervals had, in fact, captured the real arithmetic results. But the intervals were narrower than the error in the real number answers, and hence did not extend far enough to capture the 'true' answers (see results in Appendix B). In this way the interval analysis did in fact highlight the inaccuracies of the algorithm. However, these results also indicate that FFT is a numerically stable algorithm for this data. Going to double precision would probably improve accuracy of the real answers enough to have the intervals capture the 'correct' solutions. Unfortunately this conjecture cannot be tested since double precision is not available. But the conclusion regarding the stability of the FFT algorithm remains valid. Listings and output from the 8-point runs is included in Appendix B.

There was one significant change required in the benchmark driver. The program sent from WES included a calculation of the modulus of the answer. This calculation being the sum of squares of the real part (interval) and complex-part (interval) introduced a dependency error in the form of a negative argument for the square root function, INTSQT. To avoid this problem, the modulus calculation was taken out of the driver.

C. GAUSSE

The first attempts to run GAUSSE with the data provided by WES all resulted in aborts due to timer runouts. Even at a factor of 100, that is allowing the interval arithmetic run 100 times as much time as the real arithmetic run required, the program was completing only about two thirds of the problem. Since this factor alone would deter using GAUSSE in an interval arithmetic setting, it was decided to investigate alternative means for analyzing the algorithm. At this point we were unable to distinguish the causes of our problems, although it appeared that all of the difficulties described in Section II were contributing. The data sent from WES was a randomly generated set of coefficients and right-hand constants for a 100 x 100 linear system, which we discovered later was incomplete (the last card had been lost from the deck). When we reduced the problem to a 20 x 20 array of randomly generated input, the program ran to completion but produced useless results as reported in the Phase III report in March.

To gain better insight into the nature of the problems using INTERVAL with this algorithm, we decided to generate coefficients for smaller arrays which had properties known to produce 'good' results for real arithmetic GAUSS elimination with partial pivoting, and then to vary the data toward less well conditioned problems. More than fifty runs were made using coefficient arrays generated as follows:

Upper left-hand element is called IONEONE

Diagonal elements increase by 1 going down the diagonal

First super diagonal contains -10

Second super diagonal contains -5

All other elements are -1

Solution vector is all +1

By varying IONEONE between 0 and 99900, we were able to study the behavior of the algorithm while controlling the condition of the problem. Several patterns emerged quite clearly.

(1) For well conditioned matrices, the interval widths are small and actually improve (become narrower) from the last to the first computed result. There is no pivoting. See run with IONEONE = 250 in Appendix B, for example.

(2) For less well conditioned matrices, pivoting usually takes place changing the order of solution, so it is not easy to follow the accuracy of successive solutions. The pattern that consistently emerges is that the intervals become larger starting with the last variable and progressing toward the first. However, the intervals decrease in width, starting five or six values before the variable for which pivoting takes place; and then, following the pivoted variable, increase exponentially (see example with IONEONE = 2, size = 40).

(3) For randomly generated data, the interval results are worse. For example, one 20 x 20 case produced intervals of virtually $(-\infty,\infty)$ for all variables, while the best random 20 x 20 case managed only 3 or 4 place accuracy.

Regarding timing, well conditioned systems ran under INTERVAL in about 16 - 18 times the real run time using the untuned system (reported in March). With the tuned version of the system, this factor was reduced to about 13. There was significant increase in time for INTERVAL runs using randomly generated data. For example, the 20 x 20 random case took 15 times the real case run time.

IX. TUNING THE SYSTEM

A. THE AUGMENT PRIMITIVES

After studying the documentation, it was decided to rewrite the eight machine dependent primitives PACK, CCODE, MOVHOL, NUMIN, ORDER, STRCHR, STRWDS and STRLNG. These primitives had been written for the G635 at the MRC and were not in the most efficient form. In particular, use of such functions as FLD was frequent, and produced grossly inefficient object code on the G635. All eight primitives were rewritten in assembly language and installed in the package. See Appendix C.

B. LINKING

In dealing with the space-efficiency balancing problem, the size of the program AUGMENT dictates the need for overlaying it to reduce costs and space requirements. This is essential in a small memory installation but is desirable in any case. The version of the system delivered with this final report is appropriately linked for overlaying.

C. PROVISION FOR MISSING FORTRAN LIBRARY SUBROUTINES

The AUGMENT/INTERVAL package requires the availability of a specific set of subroutines in the host FORTRAN library. Honeywell's FORTRAN library for the 600 line does not include the following required routines in a readily usable form:

BPA (REAL)	EXTENDED (DOUBLE PRECISION)		
TAN	DTAN	DLOG2*	DTANH
CBRT	DARSIN	DSINH	DCERT
	CARCOS	DCOSH	DEXP2**

*Rename of DLOG
**Rename of DEXP

In order to expedite implementation, dummy routines were inserted for these eleven functions. Subsequently, Honeywell released a new FORTRAN library on their 6600 line which included all of the above routines. However, the possibility exists that these newly released routines employ instructions from E.I.S. which are not available on the WES computer.

D. MACHINE CONSTANT FRACBD

FRACBD is a machine dependent constant used in various INTERVAL routines. It is intended to provide a direct means for truncating the decimal part of a real number. The value assigned to FRACBD in the first implementation was close to but not exactly equal the number expected. As a result, there were occassional unexplained inaccuracies in the computations. The correct value has been inserted in the current version of the package. (See Appendix A).

E. MODIFICATION OF INTMUL

In the subroutine INTMUL, as a result of some questionable coding, the variable TEMP could have been used in a logical comparison before it was assigned a value. This occurred in the statement

```
IF(CASE .NE. 5 .OR. BPATMP(1) .LE. TEMP) GOTO 110
```

after the line labelled 104 and again in the statement

```
IF(CASE .NE. 5 .OR. BPATMP(1) .GE. TEMP) GOTO 120
```

after the statement labelled 112. In both cases, if TEMP had not been assigned, then CASE was not equal to 5, and so the right side of the

condition did not have to be checked. However, the Honeywell FORTRAN compiler <u>does</u> execute the code which compares BPATMP(1) and TEMP. Hence, the contents of TEMP could cause errors. By assigning zero to TEMP upon entry to INTMUL, the possibility of error is eliminated.

F. REMOVING UNNECESSARY CALLS TO BPA SUBROUTINES

Two changes were made to the description deck to increase the efficiency of the INTERVAL package.

First, all calls to BPASTR were eliminated by removing the line

SERVICE COPY (STR)

from the BPA section of the deck. The current version of INTERVAL equates the types REAL and BPA, so A = B is used now instead of CALL BPASTR(B,A).

Second, all calls to BPALT, BPALE, BPAEQ, BPANE, BPAGE, and BPAGT have been eliminated. Once again, since REAL and BPA are the same, the output source from AUGMENT uses the corresponding FORTRAN logical operators.

This was accomplished by adding an ENVIRONMENT section to the description deck, and the FORTRAN logical operators were defined for BPA operands.

To illustrate the effect of these changes, for the statement X = A * B, noted in Section II D, the number of subroutine calls is reduced from 134 to 46 and the nesting is only 2 deep.

G. CODING OF THE PRIMITIVES

This section contains a description of the algorithms developed for BPA arithmetic and for brounding, together with explanations of the design decisions made.

Initially these algorithms were written to incorporate special handling for the asymmetric numbers described below. However, after having obtained a working version which allowed the special cases to be treated correctly as far as INTERVAL was concerned, it was decided these special cases caused inefficiencies and difficulties of understanding beyond their usefulness. For this reason a change was made to the algorithms which eliminated the special cases as either inputs to or results from the arithmetic routines (there is one exception, which will be explained later). To insure accuracy during the brounding of the special cases, the original routine required the input to have the correct sign. This requirement is no longer necessary, and so the current brounding routine requires non-negative input, and the sign correction is made at the end.

Normalized, floating-point, two's complement numbers on the Honeywell 635 are not symmetric about zero. For single precision, normalized numbers the following ranges are given:

<u>negative</u> $-(1 + 2^{-26})(2^{-129}) \geq N \geq -2^{129}$

<u>positive</u> $2^{-129} \leq N \leq (1 - 2^{-27})(2^{127})$.

Furthermore, zero is represented as $(0)(2^{-128})$.

Two main problems are caused by this asymmetry: the absolute value of the negative number with the largest magnitude cannot be

represented, and the smallest, normalized, positive number has no normalized complement (although it does have an unnormalized complement). As mentioned above, these two numbers are not allowed as inputs to the arithmetic routines. Thus every valid argument has an exact complement in the INTERVAL package.

No arithmetic operation takes place if either of these two numbers is found. When the most negative number is an argument to the routines, the infinity or the overflow indicator is set, and the negative number with the largest magnitude allowed is returned as the answer. When the smallest positive number is used as an argument, the underflow indicator is set, and either zero or the smallest positive [illegible] that is allowed is returned as the answer, depending on the bro[illegible] option. The single exception to this treatment occurs in division by zero, in which the unchanged dividend is returned as the "answer," and the zero divisor indicator is set.

Understanding the following routines requires some knowledge of how the floating-point numbers are represented.

A positive, single precision, normalized, floating-point number has an 8-bit exponent and a 28-bit mantissa. The exponent part holds a two's complement integer, E, and the mantissa part holds a fractional, two's complement number, M. The relation for a floating-point number Z is this:

$$Z = M * 2^E.$$

Furthermore, the mantissa must have a zero in the first bit (the sign bit), and a 1 in the second bit. Thus if M is positive, then $2^{-1} \leq M \leq 1 - 2^{-27}$. If we catenate non-zero bits to the right end

of a positive mantissa, then the magnitude of the number represented is increased; if we truncate non-zero bits from the right end of a positive mantissa, then the magnitude is decreased.

Double precision numbers also have 8-bit exponents, but their mantissas are expanded to 64 bits. The arithmetic and brounding routines use the full AQ register to manipulate the M(A), and so a single precision mantissa occupies $M(A)_{0-27}$ with $M(A)_{28-72}$ being initially set to zero. The U register is simulated by a word pair in memory.

Given the elimination of the normalized numbers without (normalized) complements, the range of numbers allowable in the INTERVAL package is the following:

<u>negative</u> $-(1 + 2^{-26})(2^{-129}) \geq N \geq -(1 - 2^{-27})(2^{127})$

<u>positive</u> $(1 + 2^{-26})(2^{-129}) \leq N \leq (1 - 2^{-27})(2^{127})$.

Four machine dependent constants are used in the following routines.

MINNEG is the negative number with the largest magnitude.

MINPOS is the smallest positive number.

POSBND is the smallest positive number allowed in the INTERVAL package.

MAXPOS is the largest, single precision, positive number.

In addition, two addition constants are used by the brounding routine, and their mantissas are given below.

ROUND	00 00	100 00
	0 27	28 63

AUGMNT	00 01	00 00
	0 27	28 63

When one of these addition constants is used, its exponent is set to the exponent of the number to which it is being added. The mantissas of ROUND and AUGMNT are not normalized.

The arithmetic routines BPACEB, BPASUB, BPAADD, BPAMUL, and BPADIV, and the brounding routine BROUND are given below. Additional comments are given for the numbered lines.

```
BPACEB :  <<Convert the double precision A1 to single
          <<precision and bround it.
          <<RES ← A1                                   >>
     clear all indicators ;
     A ← A1 ;
     case
        : A = MINNEG :  <<Overflow.>>
             SC ← 1 ; goto .EO. ;
        : A = MINPOS :  <<Underflow.>>
             goto .EU. ;
     endcase ;
     if A < 0
        then A ← -A ;  SC ← 1 ;
     fi ;
     goto BROUND ;  <<End of BPACEB>>


BPASUB :  <<Compute RES ← A1 - A2.>>

     clear all indicators ;
     U ← A2 ;
     if U = 0
        then <<The answer is A1.>>
          A ← A1 ;
          case
             : A = MINNEG :  <<Overflow.>>
                  SC ← 1 ; goto .EO. ;
             : A = MINPOS :  <<Underflow.>>
                  goto .EU. ;
          endcase ;
          RES ← A ; return ;

     fi ;

     case
          : U = MINNEG :  <<Overflow.>>
               SC ← 1 ; goto .EO. ;
          : U = MINPOS :  <<Underflow.>>
               A ← U ; goto .EU. ;
     endcase ;
```

```
U ← -U ;
goto SUBIN ; <<BPAADD completes the subtraction.>>

BPAADD :  <<Compute RES ← A1 + A2.>>

     clear all indicators ;
     U ← A2 ;
     if U = 0
          then <<A1 is the answer.>>
          A ← A1 ;
          case
               : A = MINNEG :  <<Overflow.>>
                     SC ← 1 ; goto .EO. ;
               : A = MINPOS :  <<Underflow.>>
                     goto .EU. ;
          endcase ;
          RES ← A ; return ;

fi ;

case

     : U = MINNEG :  <<Overflow.>>
           SC ← 1 ; goto .EO. ;
     : U = MINPOS :  <<Underflow.>>
            A ← U ; goto .EU. ;
endcase ;

SUBIN :  <<This is the entry point for the subtraction routine.  >>
     A ← A1 ;
     if A = 0
          then <<U holds the exact answer.>>
               RES ← U ; return ;

fi ;

case

     : A = MINNEG :  <<Overflow.>>
           SC ← 1 ; goto .EO. ;
     : A = MINPOS :  <<Underflow.>>
           goto .EU. ;
endcase ;

if |U| < |A|
     then U ↔ A ;
fi ;
if A < 0
     then <<Make A positive.>>
          A ← -A ; U ← -U ; SC ← 1 ;
fi ;
TEMP ← E(U) - E(A) ; <<Compute shift count.>>
```

```
if TEMP > 0 <<No shift if TEMP ≤ 0.>>
     then
          E(A) ← E(U) ;
          shift M(A) right TEMP bits ; <<Line up the mantissas.>>
fi ;

A ← A + U ;
if A = 0
     then <<The answer must be exact.>>
          RES ← A ; return ;
fi ;
if RI = 1

     then <<We lost digits in the mantissa, and the
          <<number in A is too small, regardless of
          <<whether it is positive or negative.  We
          <<set bit 35 to 1; this bit is in the extra
          <<precision word, and will not affect the
          <<rounding option.                         >>

     M(A)_35 ← 1 ;

fi ;

if A < 0

     then A ← -A ; SC ← -SC ;

fi ;

goto BROUND ; <<End of BPAADD>>
```

Additional comments for BPAADD:

Lines

1 At the present time (1976-1977) the floating point compare magnitude instruction DFCMG does not work on the Honeywell 635; consequently, we program around it by making both arguments positive, and then comparing them. The problem with DFCMG was not known by Honeywell at the time it was reported to them.

2-4 The exponent of the absolute value of an exact power of two is always one greater than the exponent of its complement;

consequently, E(U) - E(A) will be negative if we are adding a power of two and its complement. If the difference in the exponents is negative or zero, then no shifting need be done. Otherwise, we need to live up the mantissas for the addition and change exponent of A accordingly. The residue indicator may be set during the shift.

5 If the computed result is MINPOS, then this negation will cause the exponent underflow indicator of the machine to be set, and then it is treated as an underflow-by-one in BROUND. The result can not be MINNEG.

```
BPAMUL :  <<Compute RES ← A1 * A2.>>
     clear all indicators ;
     U ← A2 ;
     if U = 0 or A1 = 0
          then RES ← 0 ; return ;
     fi ;
     case
          : U = MINNEG : <<Overflow.>>
                SC ← 1 ; goto .EO. ;
          : U = MINPOS : <<Underflow.>>
                 A ← U ; goto .EU. ;
     endcase ;
     if U < 0
          then <<Make it positive.>>
               U ← -U ; SC ← 1 ;
     fi ;
     A ← A1 ;
     case
```

```
        : A = MINNEG :  <<Overflow.>>
              SC ← 1 ; goto .EO. ;

        : A = MINPOS :  <<Underflow.>>
              SC ← 0 ; goto .EU. ;

        endcase ;

        if A < 0

             then <<Make it positive.>>

                  A ← -A ; SC ← -SC ;

        fi ;

1       A ← A * U ;

        goto BROUND ; <<End of BPAMUL>>
```

Additional comments for BPAMUL:

Lines

1 Single precision mantissas have 28 bits (27 bits plus a sign), and so the product can be held in 55 bits (a sign bit plus 54 for the mantissa). FMP computes the full AQ (72 bits) accurately, and so we use the machine's floating multiply instruction.

```
BPADIV : <<Compute RES ← A1/A2.>>

     clear all indicators ;

     U ← A2 ;

     if U = 0

          then <<Division by zero.>>

               BPAFLT ← 4 ; <<DZ ← 1>>

               RES ← A1 ; <<No checking here.>>

               return ;
```

```
fi ;
if A1 = 0
    then RES ← 0 ; return ;
fi ;
case
    : U = MINNEG : <<Overflow.>>
          SC ← 1 ; goto .EO. ;
    : U = MINPOS : <<Underflow.>>
          A ← U ; goto .EU. ;
endcase ;
if U < 0
    then <<Make it positive.>>
        U ← -U ; SC ← 1 ;
fi ;
A ← A1 ;
case
    : A = MINNEG :  <<Overflow.>>
          SC ← 1 ; goto .EO. ;
    : A = MINPOS ;  <<Underflow.>>
          SC ← 0 ; goto .EU. ;
endcase ;
if A < 0
    then <<Make it positive.>>
        A ← -A ; SC ← -SC ;
fi ;
V ← 0 ;
for j ← 35 downto 1 do
1       if |M(A)| ≥ |M(U)|
            then
```

```
2                   M(A) ← M(A) - M(U) ;
3                   V ← V + 1 ;

          fi ;

4         shift M(A) and V left by 1 ; <<Multiply by 2.>>

     endfor ;

     shift M(A) right 36 bits ; <<Signal remainder.>>

     M(A) ← V ; <<Move the quotient to M(A).>>

     if M(A)_0 = 1

          then <<We need to normalize the quotient.>>

               shift M(A) right 1 bit ;

               E(A) ← E(A) + 1 ; <<Correct the exponent.>>

     fi ;

5         E(A) ← E(A) - E(U) ;

6         if overflow

               then <<We may have exponent overflow or underflow.   >>

7                   if E(A)_0 = 1

                         then <<Overflow.>>

8                             goto .EO. ;

                         else <<Underflow.>>

9                             goto .EU. ;

                    fi ;

          fi ;

          if RI = 1

               then <<Set a low order bit in M(A) to 1.             >>

                    M(A)_35 ← 1 ;

          fi ;

          goto BROUND ; <<End of BPADIV.>>
```

Additional comments for BPADIV:

Lines

1-4 Yohe's algorithm for division in the base β number system includes an extra loop at this point to subtract M(U) from M(A) until M(A) < M(U). The normalized form of floating-point numbers in the Honeywell 635 makes this extra loop unnecessary; the M(A) must always be less than 2 * M(U), and so the subtraction in line 2 insures that M(A) < M(U).

5-6 If the subtraction in line 5 produces a number either too large or too small to be represented in the 8-bit exponent of the 635, then the overflow indicator of the machine is set.

7-9 To determine whether the fault was caused by exponent overflow or underflow, we inspect the exponent computed. If E(A) is too big, then the machine uses the sign bit to gain the magnitude it needs. Thus exponent overflow sets $E(A)_0$ to 1. If E(A) is too small, then $E(A)_0$ becomes zero when the subtraction is performed.

```
BROUND : <<Bround the number in A, and return the answer in RES.
         <<The brounding option is held in OPTION.  The fault
         <<value is returned in BPAFLT, and the following
         <<values are used:

         <<                0    operation successful
         <<                1    exponent overflow
         <<                2    infinity
         <<                3    exponent underflow
         <<                4    zero divisor                     >>
```

```
if OPTION ∉ {'U', 'L', 'T', 'R', 'A'}
     then <<Use R if OPTION is invalid.>>
          OPTION ← 'R' ;
fi ;
if exponent overflow or A > MAXPOS
     then goto .EO. ;
fi ;
if exponent underflow or A < POSBND
     then goto .EU. ;
fi ;
     <<The correction for the residue indicator has been made
     <<already, and is held in M(A)35.                      >>
case
1     : [(OPTION = 'U' and SC = 0) or
2     (OPTION = 'L' and SC = 1) or
3      OPTION = 'A'] and M(A)28-63 ≠ 0 :
     RES ← A + AUGMNT ;
     if exponent overflow
          then <<Set the infinity flag and load MAXPOS.>>
               BPAFLT ← 2 ;
               RES ← MAXPOS ;
     fi ;
     if SC = 1
          then RES ← -RES ;
     fi ;
     return ;
```

```
4       : (OPTION = 'U' and SC = 1) or
5         (OPTION = 'L' and SC = 0) or
6          OPTION = 'T' or M(A)28-63 = 0 :
                RES ← A ;
                if SC = 1
                     then RES ← -RES ;
                fi ;
                return
        : OPTION = 'R' :  <<the last possibility>>
                RES ← A + ROUND ;
                if exponent overflow
                     then <<set the infinity flag and load MAXPOS>>
                          BPAFLT ← 2 ;
                          RES ← MAXPOS ;
                fi ;
                if SC = 1
                     then RES ← -RES ;
                fi ;
                return ;
  end case ;

        .EO. :  <<Overflow faults are handled here.  The input
                <<number is irrelevant.                          >>
        case
              : (OPTION = 'U' and SC = 1) or
                (OPTION = 'L' and SC = 0) or
                 OPTION = 'T' :
```

```
          <<The brounding option implied truncation,
          <<and so use the fault value for exponent
          <<overflow.                                  >>

          BPAFLT ← 1 ;

     else <<Augmentation was implied, and so we had
          <<an infinity fault.                         >>

          BPAFLT ← 2 ;

endcase ;

RES ← MAXPOS ;

if SC = 1

     then RES ← -RES ;

fi ;

return ;

.EU. : <<Underflow faults are handled here.  Special
       <<action is taken if the R option is specified
       <<and we have exponent underflow by one.        >>

BPAFLT ← 3 ; <<exponent underflow>>

case

       : (OPTION = 'U' and SC = 0) or

         (OPTION = 'L' and SC = 1) or

          OPTION = 'A' :

                << The brounding option implied
                << augmentation, and so we use the
                << non-zero number with the smallest
                << magnitude allowed.                  >>

                RES ← POSBND ;

       : OPTION ≠ 'R' :

                <<Truncation was implied; so we
                <<use zero.                            >>

                RES ← 0 ;
```

```
     else <<OPTION = 'R'>>
7         if A ← POSBND or exponent underflow by 1
               then RES ← POSBND ;
               else RES ← 0 ;
          fi ;
     endcase ;
     if SC = 1
          then RES ← -RES ;
     fi ;
     return ; <<End of BROUND>>
```

Additional Comments for BROUND:

<u>Lines</u>

1-3 Augmentation is implied in 3 cases: (1) OPTION = 'U' and SC = 0 means the true answer is positive, and we want the least upper bound; (2) OPTION = 'L' and SC = 1 means the true answer is negative and we want the greatest lower bound, thus we must augment the positive number to increase the magnitude; (3) OPTION = 'A'. If $M(A)_{28-63} = 0$ then the result is exact.

4-6 Truncation is implied in 4 cases: (1) OPTION = 'U' and SC = 1 means the true result is negative, and we want the least upper bound, thus we truncate the positive result to decrease the magnitude; (2) OPTION = 'L' and SC = 0 means the true result is positive, and we want the greatest lower bound; (3) OPTION = 'T'; (4) $M(A)_{28-63} = 0$

Lines

4-6	means the result can be expressed exactly in single precision, and no brounding option will affect it, thus the extra precision bits are effectively truncated.
7	Exponent underflow-by-one can occur two ways: by computation and by input of MINPOS. All numbers smaller than POSBND and greater than or equal to MINPOS are considered to be cases of underflow by one. If we actually cause a machine exponent underflow during computation, the computed exponent is 127, or the largest exponent possible.

X. DESCRIPTION OF THE TAPE SENT TO WES

The tape sent to WES is a multifile tape, written at 800 bpi, 7 track. The fifteen files contained on the tape are:

File 1	Arithmetic and brounding routines for INTERVAL
File 2	Machine dependent primitives for AUGMENT
File 3	SESOL and BANSOL
File 4	INTERVAL source library, including corrected INTCOS and the modifications to the AUGMENT description deck to remove the superfluous subroutine calls
File 5	FFT
File 6	SPLINE(INTERVAL form)
File 7	GAUSSINT (old version of GAUSE)
File 8	GAUSS (INTERVAL form)
File 9	SESOL
File 10	FFT (original)
File 11	FFT (no complex)
File 12	FFT (INTERVAL)
File 13	FFT (8 point real)
File 14	FFT (8 point INTERVAL)
File 15	GAUSS (a third version)

XI. CONCLUSIONS AND RECOMMENDATIONS

In my opinion, the most significant single fact that has emerged from this project is that interval arithmetic has limited value as a tool for analyzing real algorithms. The limitation is specifically dependency. As pointed out by our results and, in a more general way, by Ris [2], the fact that interval arithmetic gives intervals which are too wide does not tell you, necessarily, that the real algorithm is bad nor that the real algorithm is necessarily sensitive to the real data used. In particular, it is well known that Gauss elimination with partial pivoting is a stable solution technique in a real number setting provided one is careful about conditioning and error control. What is not so well known is that the same Gauss method is subject to severe dependency problems which can cause as much as fourfold increases in interval widths at each stage of the procedure when employing interval arithmetic. The point is that interval analysis yields information about interval algorithms in all cases; but interval analysis yields information about the real version of an interval algorithm only when the interval algorithm is 'good'. One needs a way to break down the sources of increased width when the computed intervals are too wide in order to be able to say anything about the corresponding real algorithm. Even then, the conclusion may be that nothing can be inferred from the interval analysis. Both Ris [2] and Hansen [1] attack this problem, but from different points of view. Hansen proposes a generalized interval arithmetic (g.i.a.) to reduce the effect of dependency. By standardizing the form of interval representation and redefining the arithmetic operators, he is able to reduce the inherent lack of sharpness (caused by dependency) to a second order effect. Ris tackles the problem in a

different way, by defining predicates and functions on intervals which have predictable propagation properties in interval arithmetic operations. He then applies these ideas to analyze interval algorithms, highlighting those characteristics which might be used to predict the success or failure of the algorithm. Hansen's approach is more in keeping with the current project in that it reduces the factor (dependency) that limits the value of interval analysis of real algorithms. Ris' approach is designed to develop new algorithms for which interval arithmetic is well suited. Of course, once a 'good' interval algorithm is produced, it will be a 'good' real algorithm.

Getting more specific, the results of the benchmark runs indicate the following:

SPLINE is reasonably stable for the data run

FFT is very stable but suffers from loss of accuracy in the calculations for the 4096 point case.

GAUSSE is (predictably) stable for well conditioned matrices and becomes less so as the condition of the matrix deteriorates. For randomly generated data, the algorithm is less efficient and less accurate. Pivoting greatly increases widths due to dependency.

Recommendations:

In view of the number and range of difficulties encountered in attempting to use this AUGMENT/INTERVAL package for analyzing algorithms, it clearly should be treated as an experimental tool, and not considered for routine or production use in its present form.

The magnitude and complexity of many of the problems being solved at WES makes interval analysis of the solution algorithms and sensitivity analysis of the data extremely difficult to accomplish with very high reliability using the current package. A major effort to employ interval

arithmetic might better be directed toward development of new algorithms based on interval concepts (as implied in Ris' work) than to attempt analysis of the real algorithms currently in use. Alternatively, redefining the arithmetic operations and interval representation (as Hansen suggests) might convert INTERVAL into a more practical tool for analysis of the real algorithms.

REFERENCES

[1] Yohe, J. M. "Guide to Implementation of the Interval II Package on Other Hardware," Draft report, Jun 1976, Mathematics Research Center, The University of Wisconsin-Madison.

[2] Abramovitz, M. and Stegun, I. A., Handbook of Mathematical Functions, AMS 55, National Bureau of Standards, 1964.

[3] National Bureau of Standards, Tables of the Exponential Function, 1939.

[4] National Bureau of Standards, Tables of Natural Logarithms, Vol. III, 1941.

[5] National Bureau of Standards, Tables of Circular and Hyperbolic Sines and Cosines, 1940.

[6] National Bureau of Standards, Tables of Circular and Hyperbolic Tangents and Cotangents, 1947.

[7] Crary, F. D., "Language Extensions and Precompilers," Technical Summary Report No. 1317, 1973, Mathematics Research Center, The University of Wisconsin-Madison.

[8] Yohe, J. M., "Best Possible Floating Point Arithmetic," Technical Summary Report No. 1054, 1970, Mathematics Research Center, The University of Wisconsin-Madison.

[9] Crary, F. D., "The AUGMENT Precompiler; I: User Information," Technical Summary Report No. 1469, 1974 (revised 1976), Mathematics Research Center, The University of Wisconsin-Madison.

[10] Crary, F. D., "The AUGMENT Precompiler; II: Technical Documentation," Technical Summary Report No. 1470, 1975, Mathematics Research Center, The University of Wisconsin-Madison.

[11] Ladner, T. D. and Yohe, J. M., "An Interval Arithmetic Package for the UNIVAC 1108," Technical Summary Report No. 1055, 1970, Mathematics Research Center, The University of Wisconsin-Madison.

[12] Hansen, E. R., "A Generalized Interval Arithmetic," Interval Mathematics, No. 29 in the series Lecture Notes in Computer Science, Springer-Verlag, 1975.

[13] Ris, F. N., "Tools for the Analysis of Interval Arithmetic," Interval Mathematics, No. 29 in the series Lecture Notes in Computer Science, Springer-Verlag, 1975.

The following are additional computer listings and runs available from the Automatic Data Processing Center, U. S. Army Engineer Waterways Experiment Station, P. O. Box 631, Vicksburg, Miss. 39180:

Interval Primitives written at the University of Kansas
Augment Primitives written at the University of Kansas

Test programs and their results:

Summation of first 129 terms of (1/X)**(I-1)
Roots of a quadratic equation
635 rounding errors in addition
Test mathematical functions using INTRAP
Gaussian elimination routine-interval extension
Table of factorials and their natural logarithms
SPLINE (real and interval)
FFT (8-point, real arc interval)
FFT (4096-point, real and interval)
Gaussian (20X20, real and interval)

APPENDIX A: MACHINE DEPENDENT

CONSTANTS FOR 'INTERVAL II'

The following constants, in BPA format, have been provided and are all located in COMMON /BPACON/

NAME	VALUE	DESCRIPTION
PIO2I	Ø002622077325	Left endpoint of interval containing PI/2
PIO2	Ø002622077326	Right endpoint of same
PII	Ø004622077325	Left endpoint of interval containing PI
PI	Ø004622077326	Right endpoint of same
TPII	Ø006622077325	Left endpoint of interval containing 2*PI
TPI	Ø006622077326	Right endpoint of same
THPI	Ø010455457440	Right endpoint of interval containing 3*PI
ZRO	0.0	zero
ONE	1.0	1
ONM	-1.0	-1
TWO	2.0	2
BPAMNB	Ø400400000000	smallest positive BPA number (normalized)
BPAMXB	Ø376777777777	largest positive BPA number
EARGMX (EXPMXA)	88.028	largest BPA number x such that exp(s) does not overflow
EARGMN (EXPMNA)	-88.028	smallest BPA number such that exp(x) does not underflow
LNACC	.0005	half-length of interval about 1 where LN accuracy decreases
LOGACC (LGACC)	.0005	same for LOG10
SNHACC	0.5E-10	same for SINH, except interval is about 0.
TNHACC	Ø400400000001	same for TANH
MAXINT	Ø106777777777	BPA representation of largest FORTRAN integer
FRACBD	Ø066575360400	Smallest positive BPA number such that the low-order digit immediately precedes the radix point.

APPENDIX B: PRIMITIVES WRITTEN AT THE UNIVERSITY OF KANSAS

```
$       GMAP
        SYMDEF  BPAADD,BPASUB,BPADIV,BPAMUL,BPACEB
        BLOCK   BPACOM
OPTION  BSS     1                 THIS IS FOR BROUNDING OPTION
BPAFLT  BSS     7                 THIS WILL RETURN ERROR CODE
        USE
 RSSA   BBSS    8
        ERLK
SHIFT   BSS     1
TEST1   EBSS    2
TEST2   BSS     2
*                                 OKTAB CONTROLS BRANCHING IN BROUND.
*                                 UPPER HALVES OF THE WORDS HOLD BRANCH
*                                 VALUES FOR THE POSITIVE NUMBERS.
*                                 THE LOWER HALVES HOLD JUMP VALUES
*                                 FOR THE NEGATIVE NUMBERS.
OKTAB   OCT     0                 THE FIRST ENTRY IS EMPTY
        OCT     0                 U(+) = 00, U(-) = 00  IN DECIMAL
        OCT     000030000030      L(+) = 24, L(-) = 24  IN DECIMAL
        OCT     000030000000      T(+) = 24, T(-) = 00  IN DECIMAL
        OCT     000013000022      R(+) = 11, R(-) = 18  IN DECIMAL
        OCT     000000000030      A(+) = 00, A(-) = 24  IN DECIMAL
*                                 EOTAB IS A TABLE WHICH HOLDS THE CORRECT
*                                 FAULTS FOR OVERFLOW BROUNDING.  VALUES
*                                 FOR POSITIVE NUMBERS ARE IN THE UPPER
*                                 HALVES OF THE WORDS, AND THE VALUES FOR
*                                 NEGATIVE NUMBERS ARE IN THE LOWER
*                                 HALVES. 1 IS OVERFLOW. 2 IS INFINITY.
EOTAB   OCT     0                 FIRST ENTRY IS EMPTY
        OCT     000002000001      U(+) = INF, U(-) = OV
        OCT     000001000002      L(+) =  OV, L(-) = INF
        OCT     000001000001      T(+) =  OV, T(-) = OV
        OCT     000002000002      R(+) = INF, R(-) = INF
        OCT     000002000002      A(+) = INF, A(-) = INF
*                                 EUTABP AND EUTABN ARE TABLES WHICH
*                                 HOLD THE CORRECT ANSWERS FOR POSITIVE
*                                 AND NEGATIVE UNDERFLOW RESPECTIVELY.
*                                 EU STANDS FOR EXPONENT UNDERFLOW.
EUTABP  OCT     0                 FIRST ENTRY IS EMPTY
        OCT     400400000001      U(+) = POSBND
        OCT     400000000000      L(+) = ZERO
        OCT     400000000000      T(+) = ZERO
        OCT     400000000000      R(+) = ZERO EXCEPT FOR EU BY 1
        OCT     400400000001      A(+) = POSBND
EUTABN  OCT     0                 FIRST ENTRY IS EMPTY
        OCT     400000000000      U(-) = ZERO
        OCT     401377777777      L(-) = MAXNEG
        OCT     400000000000      T(-) = ZERO
        OCT     400000000000      R(-) = ZERO EXCEPT FOR EU BY 1
        OCT     401377777777      A(-) = MAXNEG
ZERO    EOCT    400000000000      ZERO IS 0*(2** -128)
        OCT     0                 MANTISSA OF DBL PREC IS 0. THIS IS ZERO
```

```
CHKDBL  OCT     000000000377     USED TO CHECK FOR ONES IS DBLE
        OCT     777777777777     PRECISION PART OF MANTISSA
MAXPOS  OCT     376777777777     EXP IS 127(LARGEST POSS.) MANTISSA
        OCT     777777777777     IS LARGEST POSITIVE FRACTION.
MINPOS  OCT     400400000000     EXP IS -128, SO THIS IS MIN POS.
        OCT     0
POSBND  OCT     400400000001     SMALLEST POSITIVE NUMBER ALLOWED
        OCT     0
MAXNEG  OCT     401377777777     EXP IS -123 AND MANT IS
        OCT     777777777777     SMALLEST NEG.
MINNEG  OCT     377000000000     EXP IS 127 AND MAN IS NEG, NORMALIZED
        OCT     0                MINNEG NOT ALLOWED AS ARGUMENT OF ANSWER
NEGBND  OCT     377000000001     NEGBND IS SINGLE PREC. NEG OF MAXPOS
        OCT     0                AND IT IS THE MOST NEGATIVE NO. ALLOWED
UTA     OCT     0                USED FOR U,T AND A OPTIONS
        OCT     777777777777
POSRND  OCT     0                ADD CONST FOR R+
        OCT     400000000000
NEGRND  OCT     0                ADD CONST FOR R-
        OCT     377777777777
EUONE   OCT     376000000000     MAX EXPONENT, UNDERFLOW BY 1
RSTABL  OCT     777,1777,3777,7777,17777,37777,77777,177777,377777
        OCT     777777,1777777,3777777,7777777,17777777,37777777,77777777
        OCT     177777777,377777777,777777777,1777777777,3777777777
        OCT     7777777777,17777777777,37777777777,77777777777
        OCT     177777777777,377777777777,777777777777
ADDONEEOCT      0
        OCT     001000000000
SUBONEEOCT      0
        OCT     1
BIGCHK  OCT     376777777777     USED TO TEST FOR OVERFLOW
        OCT     0                AGAINST MAXPOS.
EUOOFF  OCT     747777
EOON    OCT     020000           EXPONENT OVERFLOW IS BIT 22
EUON    OCT     010000           EXPONENT UNDERFLOW IS BIT 23
OFOMSK  OCT     000000004000     BIT 24 IS THE OVERFLOW MASK INDICATOR
TEMP    EBSS    2
ARG1    EBSS    2
ARG2    BSS     2
COUNT   BSS     1
SAVEXP  BSS     1
SAFE    EBSS    2
R.U.    BSS     2
ANS     BSS     2
BITS    OCT     1000000,2000000,4000000,10000000,20000000,40000000
        OCT     100000000,200000000,400000000,1000000000,2000000000
        OCT     4000000000,10000000000,20000000000,40000000000
        OCT     100000000000,200000000000,400000000000
CALL    BSS     1                USED TO IDENTIFY CALLING ROUTINE
TURN    MACRO
        STQ     TEMP+1
        LDQ     #1
```

```
        STI     TEMP
        ORSQ    TEMP
        LDI     TEMP
        ENDM    TURN
ENTER   MACRO
        SREG    .RSSA
        STI     .E.L..
        STX1    .E.L..
        LDA     OFOMSK       WE WANT TO TURN OFF THE FAULT TRAPS
        STI     TEMP         GET A COPY OF THE INDICATORS
        ORSA    TEMP         SET OVERFLOW MASK INDICATOR OF COPY
        LDA     EUOOFF       TURN EXPONENT OFLO AND UNFLO OFF
        ANSA    TEMP
        LDI     TEMP         FAULT TRAPS ARE NOW DISABLED
        LDX2    0,DU         CLEAR ERROR REGS.
        LDX4    0,DU
        LDX6    0,DU         CLEAR SC
        LDX7    0,DU         CLEAR RI
        EAX0    2,1*         GET FIRST ARG
        IFE     #1,1,21      IF ADD,SUB,MUL,OR DIV, DO NEXT 21 LINES
        FLD     0,0
        FCMP    MINNEG       MINNEG IS NOT ALLOWED AS AN ARGUMENT
        TNZ     *+2
        TRA     BROUND       AND GO TO BROUND PART .EO.
        FCMP    MINPOS       MINPOS IS NOT ALLOWED AS AN ARG
        TNZ     *+3          IF NOT MINPOS, GO ON
        LDX2    3,DU         SET UNDERFLOW FAULT
        TRA     BROUND       GO TO BROUND(.EU.) WITH UNFLO BY 1
        FST     ARG1         STORE IN DOUBLE PRECISION AREA
        STZ     ARG1+1       CLEAR EXTRA PRECISION
        EAX0    3,1*
        FLD     0,0
        FCMP    MINNEG       MINNEG IS NOT ALLOWED AS AN ARGUMENT
        TNZ     *+2          GO ON IF ARG. IS NOT MINNEG
        TRA     BROUND       AND GO TO BROUND PART .EO.
        FCMP    MINPOS       MINPOS IS NOT ALLOWED AS AN ARGUMENT
        TNZ     *+3          GO ON IF NOT MINPOS
        LDX2    3,DU         SET UNDERFLOW FAULT
        TRA     BROUND       GO TO BROUND(.EU.) WITH EU BY ONE
        FST     ARG2         SAME FOR ARG2
        STZ     ARG2+1       CLEAR EXTRA PRECISION.
        IFE     #1,2,4       IF BPACEB, DO NEXT FOUR LINES
        LDA     0,0
        LDQ     1,0
        STA     ARG1
        STQ     ARG1+1
        LDA     #1,DL        STORE CALL TYPE IN CALL. CEB=2, OTHERS=1
        STA     CALL
        ENDM    ENTER
ARSHFT  MACRO
        STX0    TEMP
        LDX0    COUNT        GET SHIFT COUNT AND SKIP IF ZERO
```

```
        TZE     *+14
        CMPX0   37,DU          SHIFT NORMAL IF .LE. 36 BIT SHIFT
        TMI     *+11
        CMPX0   63,DU          SHIFT .GE. 63, CHECK FOR ONES
        TMI     *+6
        CANA    =O777777777777 CHECK FOR ANY ONES
        TZE     *+2            IF NOT THEN SKIP, ELSE GO ON
        LDX7    1,DU           SET RESIDUE INDICATOR
        DFLD    ZERO           ALL BITS WERE SHIFTED OUT SO LOAD ZERO
        TRA     *+5            NOW PREPARE TO RETURN
        CANA    RSTABL-37,0    CHECK FOR ONES THAT WILL BE LOST
        TZE     *+2            IF NONE THEN GO SHIFT
        LDX7    1,DU           ELSE SET RESIDUE INDICATOR
        LRL     0,0            SHIFT MANTISSA RIGHT COUNT BITS
        LDX0    TEMP
        ENDM    ARSHFT
NEGU    MACRO
        DFST    SAFE
        DFLD    R.U.
        FNEG
        DFST    R.U.
        DFLD    SAFE
        ENDM    NEGU
BPACEB  ENTER   2              ENTRY TO CONVERT EXTENDED TO BPA
        DFLD    ARG1
        TRA     BROUND
BPAMUL  ENTER   1
        FLD     ARG1           MAKE ARGS POSITIVE IF NEC.
        TPL     BPAM.1         SET SC IF WE CHANGE SIGN
        FNEG
        FST     ARG1
        LDX6    1,DU
BPAM.1  FLD     ARG2
        TPL     BPAM.3
        FNEG
        FST     ARG2
        CMPX6   1,DU           NEED TO FLIP SC.
        TZE     BPAM.2
        LDX6    1,DU
        TRA     BPAM.3
BPAM.2  LDX6    0,DU
BPAM.3  FLD     ARG1           MULTIPLY, DO NOT NORMALIZE
        UFM     ARG2
        TEU     BPAMEU
        TEO     BPAMEO
        FNO                    EXPONENT IS OK, TRY TO NORMALIZE.
        TEU     BPAMEU         WE CAN ONLY MAKE THE EXPONENT SMALLER.
        CMPX6   0,DU           GET SIGN ANSWER CORRECT.
        TZE     BROUND
        FNEG
        TRA     BROUND
BPAMEU  FNO                    GET THE ANSWER IN THE BEST FORM POSS.
```

```
        LXL4    OPTION
        STZ     BPAFLT
        CMPX6   0,DU
        TZE     .EU.            BEFORE TRANSFER, MAKE SURE ANS IS POS.
        FNEG
        TRA     .EU.
BPAMEO  LXL4    OPTION
        STZ     BPAFLT
        DFST    ANS             SAVE THE RESULT AND SEE HOW MUCH WE MUST
        LDE     0,DU            SHIFT TO NORMALIZE. R(E) WILL HAVE NEG,
*  NEGATE BEFORE BROUND IF NEEDED
        CMPX6   0,DU
        TZE     BPAM.4
        FNEG
BPAM.4  FNO                     SHIFT NUM. IT MAY BE ENOUGH TO FIX THE
        ADE     ARG1            OVERFLOW. THIS FIRST ADD CAN'T OVERFLOW.
        ADE     ARG2            BECAUSE BOTH EXP. OF VAR. ARE POSITIVE
        TEO     .EO.
        TRA     BROUND
BPADIV  ENTER   1
        FLD     ARG1            MAKE BOTH MANTISSA POSITIVE
        TPL     BPAV.1
        FNEG
        DFST    ARG1
        LDX6    1,DU
BPAV.1  FLD     ARG2
        TZE     ZERODV
        TPL     BPAV.3
        FNEG
        DFST    ARG2
        CMPX6   1,DU
        TZE     BPAV.2
        LDX6    1,DU
        TRA     BPAV.3
BPAV.2  LDX6    0,DU
BPAV.3  STZ     SHIFT
        FLD     ARG2            SEE IF DIVIDEND IS LARGER THAN DIVISOR
        STA     TEMP            THIS WORKS IF BOTH ARGS ARE NORMALIZED
        FLD     ARG1
        CMPA    TEMP
        TMI     BPAV.5
        TNZ     *+5             IF THE MANTISSAS ARE .5 AND THE SIGN
        CMPX6   0,DU            WAS CHANGED, THEN WE MUST NOT ADD
        TZE     *+3             ONE TO THE SHIFT COUNT
        CMPA    =O200000000000  MANTISSA OF ONE HALF
        TZE     *+2
        AOS     SHIFT
        LRS     1
        TRA     BPAV.4
BPAV.5  FLD     ARG1
BPAV.4  DVF     TEMP
        CMPQ    0,DU
```

```
        TZE     SMALL
        LDX7    1,DU              FILL RESIDUE IND IF THERE IS REMAINDER.
SMALL   CMPX6   1,DU              SEE IF ANS IS NEG.
        TNZ     EXP
        LDQ     0,DU
        LDE     0,DU              CLEAR EXPONENT TO PREVENT ERROR IN NEG
        FNEG
EXP     LDE     0,DU
        LDQ     SHIFT
        CMPQ    0,DU
        TZE     EXP.1
        LDQ     0,DU
        LDE     =0002000,DU
                 * BACKSPACE
        LDE     =0002000,DU
EXP.1   FNO
        DFST    ANS
        LDA     ARG1
        ANA     =0776000,DU
        ARS     28
        LDQ     ARG2
        ANQ     =0776000,DU
        QRS     28
        STQ     TEMP
        SBA     TEMP
        LDQ     ANS
        ANQ     =0776000,DU
        QRS     28
        STQ     TEMP
        ADA     TEMP
        CMPA    =128
        TPL     EXPO              CHECK IF EXP IN RANGE
        CMPA    =-128
        TMI     EXPU
        ALS     28
        STA     TEMP
        DFLD    ANS
        LDE     TEMP
        DFST    ANS
        CMPX7   1,DU              IF THERE WAS A REMAINDER, ADD ONE
        TNZ     BROUND
        STE     ADDONE
        FLD     ADDONE
        TRA     BROUND
EXPO    LXL4    OPTION
        STZ     BPAFLT
        TRA     .EO.
EXPU    LXL4    OPTION
        STZ     BPAFLT
        TRA     .EU.
ZERODV  LDA     4,DL              DIVISION BY ZERO CASE
        STA     BPAFLT
        TRA     EXIT
BPAADD  ENTER   1
```

```
        DFLD    ARG2          GET ARG2, TEST FOR ZERO, STORE IN R.U.
        TNZ     *+3
        FLD     ARG1          ARG2 = 0, LOAD ARG1 AND RETURN
        TRA     EXIT
        DFST    R.U.
SUBIN   DFLD    ARG1          BPASUB COMES HERE. GET ARG1.
        TNZ     ADD1          IF ARG1 = 0, PREPARE TO LEAVE BPAADD
        DFLD    R.U.          OUR ANSWER IS ARG2
        TRA     EXIT          GO TO EXIT, ONE ARG WAS ZERO
ADD1    DFST    TEST1
        DFLD    R.U.          WE WILL MAKE BOTH POSITIVE AND COMPARE
        TPL     ADD2          DFCMG DOES NOT WORK (NOV.1976)
        FNEG                  MAKE 2ND ARG POSITIVE FOR DFCMP
ADD2    DFST    TEST2         NOW RELOAD TEST1 AND MAKE +
        DFLD    TEST1
        TPL     ADD3
        FNEG                  MAKE R(EAQ) +
ADD3    DFCMP   TEST2         MAKE THE COMPARISON
        TZE     ADD5          IF EQUAL, DO NOT SWITCH
        TMI     ADD5          IF R(EAQ) .LT. R.U., LEAVE
ADD4    DFLD    ARG1          R(EAQ) MAY BE WRONG SIGN, SO RELOAD ARG1
        DFST    R.U.          AND SWITCH R(EAQ) AND R.U.
        DFLD    ARG2
        TRA     ADD6
ADD5    DFLD    ARG1          RECONSTITUTE THE SIGN OF R(EAQ)
ADD6    DFCMP   ZERO          R(EAQ) MUST BE SET POSITIVE
        TPL     ADD7
        FNEG                  NEGATE R(EAQ) AND R.U., SET SIGN CHANGE
        NEGU
        LDX6    1,DU
ADD7    STE     SAVEXP        GET EXPONENTS, COMPUTE SHIFT COUNT
        LDX0    R.U.
        ANX0    =0776000,DU
        SBX0    SAVEXP
        TOV     *+2           IF OVERFLOW, THE SHIFT NUMBER
        TRA     *+2           WAS TOO BIG, AND SO WE LOAD 64
        LDX0    =0200000,DU   AS THE SHIFT COUNT
        TMI     ADD8          IF SHIFT IS NEGATIVE NO SHIFT NEEDED
        TZE     ADD8          NO SHIFT NEEDED, SHIFT COUNT WAS ZERO
        EAQ     0,0
        QRL     10
        STQ     COUNT         STORE SHIFT COUNT IN COUNT
        LDQ     =0
        ARSHFT  COUNT         SHIFT MANTISSA RIGHT COUNT BITS
        LDE     R.U.          LOAD CORRECT EXPONENT FOR R(EAQ)
ADD8    DFAD    R.U.          ADD THE TWO NUMBERS AND NORMALIZE.
        CMPX6   0,DU          TEST FOR SIGN CORRECTION
        TZE     ADD11         AND SKIP TO ADD11 IF NONE NEEDED
        TEO     *+3           GO TO CORRECTION FOR EXPONENT OFLO
        TEU     ADD9          SAME FOR EXPONENT UNDERFLOW
        TRA     ADD10         AND AGAIN IF NO FAULT OCCURRED
*                             MINPOS IS THE RESULT OF ONE OVERFLOW
```

```
*                                   CORRECT NEGATIVE IS MINNEG, BUT NEGATIVE
*                                   MINPOS GIVES UNDERFLOW. MINNEG IS NOT
*                                   ALLOWED IN INTERVAL, SO GIVE IT OVERFLOW
*                                   TEU TURN OFF UNDERFLOW IF WE NEGATED
*                                   MINPOS.
         FNEG                       NEGATE THE MANTISSA
         TEU      *+1
         TURN     EOON              TURN EO BACK ON
         TRA      ADD11             WE ARE THROUGH NEGATING OVERFLOW
*                                   MINNEG IS THE RESULT OF ONE UNDERFLOW;
*                                   CORRECT NEGATIVE IS MINPOS, BUT NEGATIVE
*                                   MINNEG GIVES OVERFLOW.  MINPOS IS NOT
*                                   ALLOWED IN INTERVAL, SO GIVE IT UNFLOW.
*                                   TEO TURNS THE OVERFLOW INDICATOR OFF IF
*                                   WE NEGATED MINNEG.
ADD9     FNEG                       NEGATE THE UNDERFLOW ANSWER
         TEO      *+1               TURN OFLO OFF IF IT IS ON
         TURN     EUON              TURN EU INDICATOR BACK ON
         TRA      ADD11             AND GO TO ADD11
ADD10    FNEG                       NO PROBLEM. JUST NEGATE
ADD11    CMPX7    0,DU              CHECK RESIDUE INDICATOR, EXIT IF ZERO
         TNZ      ADD12
         DFCMP    ZERO              EXIT IF ZERO, ELSE GO TO BROUND
         TZE      EXIT
         TRA      BROUND
*
*
*        AT THIS POINT WE MAY NEED A RESIDUE CORRECTION.
*        WE HAVE THE FOUR FOLLOWING POSSIBLE CASES..
*                 1. SC = 0 AND R(EAQ) .GE. 0
*                 2. SC = 0 AND R(EAQ) .LT. 0
*                 3. SC = 1 AND R(EAQ) .GE. 0
*                 4. SC = 1 AND R(EAQ) .LT. 0
*
*        CASES 1 AND 2.  IF THE EXTRA PRECISION WORD
*                        IS 0, SET IT TO 1,
*                        AND THEN GO TO BROUND.
*        CASES 3 AND 4.  IF THE EXTRA PRECISION WORD
*                        IS 0, SUBTRACT 1 FROM THE
*                        FULL MANTISSA, AND THEN GO
*                        TO BROUND.
*
*
ADD12    CMPX6    0,DU              IDENTIFY THE CASE. CHECK SC
         TNZ      ADD13             IF SC = 1, GO TO ADD13
         DFST     TEMP
         LDQ      TEMP+1
         TNZ      BROUND            IF Q NOT ZERO, WE ARE DONE HERE
         LDQ      =1
         STQ      TEMP+1            SET Q TO 1 REPLACE IN TEMP
         DFLD     TEMP              LOAD TEMP AND GO TO BROUND
         TRA      BROUND
```

```
ADD13  DFST    TEMP        WE HAVE CASE 3 OR 4
       LDQ     TEMP+1      CHECK FOR NONZERO AND LEAVE IF SO
       TNZ     BROUND
       DFLD    TEMP        ELSE SUBTRACT ONE FROM AQ
       STE     SUBONE
       DFSB    SUBONE
       TRA     BROUND
BPASUB ENTER   1
       DFLD    ARG2        LOAD ARG2, TEST FOR ZERO, NEGATE
       TNZ     *+3
       FLD     ARG1
       TRA     EXIT
       FNEG                NEGATE AND GO TO  SUBIN IN BPAADD
       DFST    R.U.
       DFST    ARG2
       TRA     SUBIN
BROUND FNO                 NORMALIZE THE R(EAQ) IF IT IS NOT
       STZ     BPAFLT      PREPARE RETURN VARIABLE
       DFCMP   ZERO        IF ANSWER IS ZERO WE ARE DONE
       TZE     EXIT        THEN GO TO EXIT,WE ARE DONE
       LXL4    OPTION
       TMI     *+4
       TZE     *+3
       CMPX4   6,DU
       TMI     *+2
       LDX4    4,DU
       TEO     .EO.        DID OP CAUSE EXP OVERFLOW OR UNDERF.
       TEU     .EU.
       DFCMP   MINPOS      IF ANSWER IS MINPOS, GO TO .EU.1
       TZE     .EU.1
       DFCMP   NEGBND      IF ANS.>=NEGBND,NO ERROR
       TMI     .EO.
       DFCMP   BIGCHK      IF BIGGER,THEN OFLOW EVEN THOUGH
       TMI     *+3         IT IS SMALLER THAN MAXPOS
       TZE     BRND.1      GO LOAD OKTAB JUMP VALUE
       TRA     .EO.        WE HAVE OVERFLOW, WITHOUT EO ON.
       CANAQ   CHKDBL      IF NO ONES IN DBL PREC PART OF MANT
       TZE     EXIT        THEN WE NEED NO BROUNDING
       DFCMP   ZERO        CHOOSE UPPER OR LOWER HALF OF WORD
       TMI     *+3         TAKE LOWER HALF IF MINUS
BRND.1 LDX4    OKTAB,4     GET JUMP VALUE FROM TABLE
       TRA     NOFLT,4     GO TO RIGHT BROUNDING
       LXL4    OKTAB,4     GET JUMP VALUE FROM TABLE
       TRA     NOFLT,4     GO TO BROUNDING FOR MINUS
.EO.   FCMP    ZERO        TEST WHETHER POSITIVE OR NEGATIVE
       TMI     .EO.1       IF NEGATIVE GO TO .EO.1
       LDX2    EOTAB,4     LOAD FAULT VECTOR FROM TABLE
       FLD     MAXPOS      LOAD POSITIVE ANSWER
       TRA     EXIT
.EO.1  FLD     NEGBND      LOAD NEGATIVE ANSWER
       LXL2    EOTAB,4     LOAD FAULT VECTOR
       TRA     EXIT        WE ARE THROUGH
```

```
.EU     LDX2    3,DU            LOAD UNDERFLOW FOR ALL OPTIONS
        FCMP    ZERO            TEST IF POSITIVE OR NEGATIVE
        TMI     .EU.2           IF NEGATIVE, GO TO .EU.2
        CMPX4   4,DU            ROUNDING TAKES SPECIAL TREATMENT
        TZE     *+3             SO SKIP THESE TWO LINES
        FLD     EUTABP,4        LOAD CORRECT ANSWER FOR U,L,T,A
        TRA     EXIT            AND GO TO EXIT
        STE     SAVEXP          CHECK IF UNFLO BY 1
        LDX0    SAVEXP          R(EAQ) NOW HAS ZERO FOR R OPTION
        CMPX0   EUONE           IF UNFLO BY 1, LOAD POSBND
        TNZ     *+3
.EU.1   FLD     POSBND
        TRA     EXIT            WE ARE DONE WITH POSITIVE CASE
        FLD     ZERO            IF EU BY >1 LOAD ZERO
        TRA     EXIT            AND GO TO EXIT
.EU.2   CMPX4   4,DU            IF NOT ROUND, LOAD VALUE FROM TABLE
        TZE     *+3
        FLD     EUTABN,4        LOAD RIGHT ANSWER FOR U,L,T,A
        TRA     EXIT            AND GO TO EXIT
        STE     SAVEXP          CHECK IF UNFLO BY 1
        LDX0    SAVEXP          R(EAQ) NOW HAS ZERO FOR R OPTION
        CMPX0   EUONE           IF UNFLO BY 1, LOAD MAXNEG
        TNZ     *+3             ELSE GO LOAD ZERO
        FLD     MAXNEG
        TRA     EXIT
        FLD     ZERO            LOAD ZERO AS THE ANSWER, EU > 1
        TRA     EXIT            AND GO TO EXIT
*                               THE FIRST BROUNDING WE DO IS U(+,-),
*                               T(-), AND A(+).  IN ALL CASES, WE
*                               ADD UTA, WHICH GENERATES A CARRY
*                               INTO THE SINGLE PREC. ANSWER IF THERE
*                               ARE ANY 1'S IN THE EXTRA PREC.WORD
NOFLT   STE     UTA
        DFAD    UTA
        TEO     *+3             OFLO IS POSITIVE. HAVE U OR A
        TEU     *+5             UNDERFLOW IS NEGATIVE, T
        TRA     EXIT            GO TO EXIT IF ANSWER OK
        LDX2    2,DU            SET INFINITY FOR OVERFLOW
        DFLD    MAXPOS          WANT MAXPOS AS ANSWER
        TRA     EXIT
        LDX2    3,DU            SET UNDERFLOW
        DFLD    ZERO            LOAD ZERO ON T OPTION
        TRA     EXIT            END OF UTA BROUNDING
*                               ROUND POSITIVE NUMBER. BIT 28 MUST
*                               BE 1 TO GO UP, ELSE CHOP OFF EXTRA
*                               PRECISION.
RPOS    STE     POSRND
        DFAD    POSRND
        TEO     *+2             OVERFLOW IS POSITIVE
        TRA     EXIT
        LDX2    2,DU            SET INFINITY
        DFLD    MAXPOS          WANT MAXPOS AS ANSWER
```

```
         TRA      EXIT
*                                  ROUND A NEGATIVE ANSWER. ROUND
*                                  VALUE IN MIDDLE TOWARD ZERO.
*                                  BIT 26 MUST BE 0 TO GO AWAY
*                                  FROM ZERO.
RNEG     STE      NEGRND
         DFAD     NEGRND
         TEU      *+2              UNDERFLOW IS NEGATIVE
         TRA      EXIT
         LDX2     3,DU             SET UNDERFLOW, UNFLO BY 1
         DFLD     MAXNEG           SO LOAD MAXNEG, NOT 0
EXIT     SXL2     BPAFLT           RETURN FAULT VECTOR IN COMMON
         LXL2     CALL
         CMPX2    2,DU
         TZE      EXIT.1
         EAX0     4,1*
         FST      0,0              AND BROUNDED ANSWER IN ARGUMENT
         TRA      EXIT.2
EXIT.1   EAX0     3,1*
         FST      0,0
EXIT.2   LREG     .RSSA
         RET      .E.L..
         END
```

In accordance with letter from DAEN-RDC, DAEN-ASI dated 22 July 1977, Subject: Facsimile Catalog Cards for Laboratory Technical Publications, a facsimile catalog card in Library of Congress MARC format is reproduced below.

Arnold, James Q
Implementation and evaluation of interval arithmetic software; Report 3: The Honeywell G635 System / by James Q. Arnold, Frank P. Ford, Richard G. Hetherington, Department of Computer Science, University of Kansas, Lawrence, Kans. Vicksburg, Miss. : U. S. Waterways Experiment Station ; Springfield, Va. : available from National Technical Information Service, 1979.
74, [13] p. ; 27 cm. (Technical report - U. S. Army Engineer Waterways Experiment Station ; 0-79-1, Report 3)
Prepared for Office, Chief of Engineers, U. S. Army, Washington, D. C.
References: p. 73-74.

1. Algorithms. 2. Computer systems programs. 3. Evaluation. 4. Honeywell G635 System. 5. Interval arithmetic. I. Ford, Frank P., joint author. II. Hetherington, Richard G., joint author. III. Kansas. University. Dept. of Computer Science. IV. United States. Army. Corps of Engineers. V. Series: United States. Waterways Experiment Station, Vicksburg, Miss. Technical report : 0-79-1, Report 3.
TA7.W34 no.0-79-1 Report 3